Symbolic Verification of TLA+ Specifications with Applications to Distributed Algorithms

DISSERTATION
submitted in partial fulfillment of the requirements for the degree of
Doktor der Technischen Wissenschaften

by

Dipl. Ing. Thanh Hai Tran
Registration Number 01428553

to the Faculty of Informatics
at the TU Wien
Advisor: Privatdoz. Dr. Igor Konnov
Second advisor: Privatdoz. Dr. Josef Widder

The dissertation has been reviewed by:

__________________________________________  _________________________________
Michael Leuschel                             Roopsha Samanta

Vienna, 7th July, 2023

__________________________________________
Thanh Hai Tran
Erklärung zur Verfassung der Arbeit

Dipl. Ing. Thanh Hai Tran

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, 7. Juli 2023

________________________

Thanh Hai Tran
Acknowledgements

First and foremost, I would like to thank my supervisors, Igor Konnov and Josef Widder. Their insightful guidance, support, enthusiasm, encouragement, and patience have shaped every step of the way, and have been important in making my research journey both fruitful and enjoyable. It has been an enormous privilege to work with Igor and Josef.

To my colleague and collaborator Jure Kukovec, thank you for many interesting and supportive discussions, and for useful feedback along the way.

Many thanks to Alexey Gotsman, who hosted me during my research stay at IMDEA Software Institute, Madrid.

My sincere gratitude goes to my examination committee, in particular to the external examiners Michael Leuschel and Roopsha Samanta, who agreed to review this thesis and provided valuable feedback.

I am also grateful to Juliane Auerböck, Beatrix Buhl, Eva Nedoma, and Toni Pisjak for always making sure that I could focus on my research rather than the legal issues, paperwork, and IT infrastructure.

I am grateful for the generous financial support provided by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103) and by the Austrian Science Fund (FWF) through the Doctoral College LogiCS (W1255-N23).

Being part of the doctoral college on Logical Methods in Computer Science (LogiCS) was a great blessing. To all the LogiCS students, thank you for the nice memories we have made together during the past years. To my office mates Jure Kukovec, Marijana Lazić, Jakob Rath, and Ilina Stoilkovska, thank you for making my everyday life more entertaining.

Finally, I am extremely grateful to my family and to my fiancée Thi Bao Tran Nguyen, for her endless support, which makes it all possible.
Abstract

TLA+ is a language for formal specification of concurrent and distributed protocols. TLA+ is extremely concise yet expressive: The language primitives include Booleans, integers, functions, tuples, records, sequences, and sets thereof, which can be also nested. This is probably why the only model checker for TLA+ (called TLC) relies on explicit enumeration of values and states.

This thesis has two main parts. First, we bring symbolic verification to the specification language TLA+. Second, we focus on formal verification techniques for the partial synchrony model of distributed computations. We demonstrate our methodology and approaches to the second part by conducting a case study about the Chandra and Toueg failure detector.

Part 1. In this thesis, we first bring symbolic verification to TLA+ specifications by developing the symbolic model checker called APALACHE. Like TLC, APALACHE assumes that all specification parameters are fixed and all states are finite structures. Unlike TLC, APALACHE translates the underlying transition relation into quantifier-free SMT constraints, which allows us to exploit the power of SMT solvers. Designing this translation is the central challenge that we address in the first part of this thesis. Our experiments show that APALACHE outperforms TLC on checking inductive invariants and finding counterexamples in instances with large state spaces.

In TLA+, a specification is written as a logical formula without assignments and other imperative statements. To improve APALACHE’s performance, we introduce an automatic technique to slice a TLA+ specification in symbolic transitions, which are used as an input to APALACHE. In contrast to TLC, our technique does not explicitly evaluate expressions, but it reduces the problem of finding symbolic transitions to the satisfiability of an SMT formula.

Part 2. In the second part of this thesis, we focus on symbolic verification techniques for partial synchrony that is a model of computation in distributed algorithms and modern blockchains. In this model, correctness of algorithms requires the existence of bounds Δ on message delays and Φ on the relative speed of processes after reaching Global Stabilization Time (GST). This makes partially synchronous algorithms parametric in time bounds. Moreover, partially synchronous algorithms are parameterized in the
number of processes. Therefore, in general we cannot verify all instances of a partially synchronous distributed algorithm by applying APALACHE or other state-of-the-art model checkers, e.g., TLC, Spin, or NuSMV.

The failure detector of Chandra and Toueg is a well-known algorithm to detect crashed processes in a system. Importantly, correctness of the failure detector is based only on the existence of bounds $\Delta$ and $\Phi$ after GST. Hence, we choose the failure detector as a case study in our thesis. To verify all instances of the failure detector, we develop the two following techniques.

While the general parameterized verification is undecidable, many distributed algorithms such as mutual exclusion, cache coherence, and distributed consensus enjoy the cutoff property, which reduces the parameterized verification problem to verification of a finite number of instances. The failure detector does not fall into one of the known classes since they typically rely on point-to-point communication and timeouts. In this thesis, we formalize this communication structure and introduce the class of symmetric point-to-point algorithms. We show that the symmetric point-to-point algorithms have a cutoff. By these results, it is sufficient to verify the failure detector by checking instances with only two processes.

We do parametric verification of both safety and liveness of the failure detector in three frameworks: TLA+, counter automata, and IVy. We introduce encoding techniques and an abstraction of in-transit messages to efficiently specify the failure detector in each framework. By running the model checkers for TLA+ (TLC and APALACHE) and counter automata (FAST), we prove safety for fixed time bounds. This helps us to find the inductive invariants for fixed parameters, which we used as a starting point for the proofs with IVy. By running IVy, we prove safety for arbitrary time bounds. Moreover, we show how to verify liveness of the failure detector by reducing the verification problem to safety verification. Thus, both properties are verified by developing inductive invariants with IVy. We conjecture that correctness of other partially synchronous algorithms may be proven by following the presented methodology.
Contents

Abstract vii

Contents ix

1 Introduction 1

1.1 Motivating Example 1: Two-phase Commit in TLA+ 3
1.2 Motivating Example 2: Chandra and Toueg Failure Detector 5
1.3 Problem Statements 8
1.4 State of the Art 8

1.4.1 General-purpose Specification Languages 8
1.4.2 Model Checkers for Specialized Languages 10
1.4.3 Integration of SMT Solvers and Theorem Provers 10
1.4.4 Cutoffs 12
1.4.5 Formal Verification for Partial Synchrony Constraints 12

1.5 Methodology 13

1.5.1 Symbolic Model Checking for TLA+ 13
1.5.2 Symbolic Verification Techniques for Partial Synchrony Constraints 14
1.5.3 Case Studies and Applications in Formal Verification 15

1.6 Publications 16
1.7 Structure of the Thesis 17

I A General-purpose Symbolic Model Checker for TLA+

2 Extracting Symbolic Transitions from TLA+ specifications 21

2.1 Overview 21
2.2 Abstract Syntax α-TLA+ 23
2.3 Preliminary Definitions 25
2.4 Formalizing Symbolic Assignments 26
2.5 Finding Assignment Strategies with SMT 28

2.5.1 Homogeneous Sets 28
2.5.2 Minimal Covering Sets 29
2.5.3 Acyclic Assignments 30
2.6 Soundness of our Approach ................................ 30
2.7 Experiments and Potential Applications .......... 33
2.8 Related Work .................................................. 35
2.9 Summary ..................................................... 36

3 TLA+ Model Checking Made Symbolic ..................... 37
3.1 Our Approach at a Glance ................................ 37
3.2 Preprocessing: Flattening, Assignments, and Types .......... 39
  3.2.1 Flattening ........................................... 39
  3.2.2 Assignments and Symbolic Transitions ............. 40
  3.2.3 Types .............................................. 40
3.3 KERA+: the Kernel Language of TLA+ Expressions .......... 41
3.4 Defining TLA+ Operators in KERA+ ..................... 42
  3.4.1 Logic ............................................... 42
  3.4.2 Sets .............................................. 43
  3.4.3 Functions .......................................... 43
  3.4.4 Records and Tuples ................................ 43
  3.4.5 Control ............................................ 43
  3.4.6 Sequences .......................................... 43
3.5 Rewriting Framework ........................................ 44
  3.5.1 Cells ............................................... 44
  3.5.2 Arenas ............................................ 45
  3.5.3 SMT Constraints .................................... 46
  3.5.4 Abstract Reduction System (ARS) .................. 47
3.6 Sets .......................................................... 48
3.7 Picking Set Elements ....................................... 51
3.8 Tuples and Records ......................................... 54
3.9 Functions and Sequences .................................. 54
3.10 Control Operators and Quantifiers ....................... 56
3.11 Additional Reduction Rules from TLA+ to KERA+ .......... 58
3.12 Implementation ............................................ 59
3.13 Experiments ............................................... 61
  3.13.1 Benchmarks ...................................... 61
  3.13.2 Experiments with Inductive Invariants ........... 62
  3.13.3 Experiments with Bounded Model Checking ........ 64
  3.13.4 Discussion on Performance ......................... 65
3.14 Summary .................................................. 65

4 Soundness of the Reduction from TLA+ to SMT ............... 67
4.1 Models and extended models in KERA+ .................. 67
  4.1.1 Models in KERA+ .................................. 67
  4.1.2 Finite structures .................................... 69
  4.1.3 Extended Models in KERA+ ....................... 70
4.2 Soundness and Termination: Overview ..................... 71
7.2.1 The System with One Sender and One Receiver .......................... 159
7.2.2 Encoding the Message Buffer ................................................. 159
7.2.3 Encoding the Relative Speed of Processes ................................. 163
7.3 Reduce Liveness Properties to Safety Properties ......................... 163
  7.3.1 Eventually Strong Accuracy ............................................... 163
  7.3.2 Strong Completeness ...................................................... 164
7.4 Experiments for Small $\Delta$ and $\Phi$ ..................................... 165
  7.4.1 Model Checkers for TLA$^\ast$: TLC and APALACHE ................ 165
  7.4.2 FAST ................................................................. 169
7.5 Ivy Proofs for Parametric $\Delta$ and $\Phi$ ................................ 169
7.6 Summary .............................................................................. 171

8 Conclusions and Future Work .................................................. 173

List of Figures ........................................................................... 175

List of Tables ............................................................................ 177

List of Algorithms .................................................................... 179

Bibliography ............................................................................. 181
CHAPTER

Introduction

Since the turn of this millennium, we have been experiencing rapid development and expansion of networking, hardware technologies and software products. These factors make distributed systems more economical, effective, and reliable. Nowadays, it is not easy to see a stand-alone computer system since most systems around us such as smartphones, medical devices, and airplanes are networked and distributed. While researchers and engineers have made a lot of efforts in this area, the design of distributed systems are still difficult and extensive task.

Typical failures of real-world distributed systems are described in [BK14]. For example, while the Amazon Cloud Service EC2 was designed to work normally despite of possible failures of individual components, a faulty network configuration change disturbed the normal system behaviour for approximately 12 hours [Ama]. Ultimately, some information could not be restored into a consistent state at all.

Why are distributed systems difficult to design and verify? One important reason is the characteristics of distributed algorithms, which is the core of such systems. First, distributed algorithms contain a set of participants with limited local knowledge, asynchrony, and failures. They make distributed algorithms non-deterministic and construct an exponential number of execution scenarios [AW04]. Second, distributed algorithms are parameterized in some manner, e.g., the number of participants or the size of message buffers. Hence, to ensure the correctness of a distributed algorithm, we have to verify infinitely many of its instances. Unfortunately, the parameterized verification problem is typically undecidable, even if every participant follows the same code [AK80, Suz88, BJK+15]. Third, distributed algorithms [FLPS85, DLS88, AW04, CT96, BCG20] require time constraints to guarantee liveness properties. For example, the existence of bounds $\Delta$ on message delay, and $\Phi$ on the relative speed of processes after some time point in partial synchrony. This combination makes distributed algorithms parametric in time bounds.
1. Introduction

In addition, researchers usually discuss distributed algorithms semi-formally. For example, computing models are traditionally described in natural languages, algorithms are presented in pseudo-code, and their properties are shown with manual proofs. These problems make correctness of distributed systems more difficult to check, even for experienced algorithm designers.

In spite of the mentioned challenges, the benefits of new methods to improve the reliability and robustness of distributed systems and algorithms are worthwhile to pursue. In the last decades, researchers have developed many specification and verification techniques for distributed algorithms [Lam02, LT88, MP20, DWZ20].

Temporal Logic of Actions (TLA) [Lam94] is a long-term project for designing a general-purpose formal language to specify algorithms. As TLA was initially designed for writing mathematical proofs about algorithms, it did not offer a concrete syntax for their specifications. Rather, the algorithm designers were expected to present their algorithms in first-order logic and choose a convenient interpretation. This gap was closed with the introduction of TLA+ [Lam02], which offers a rich syntax for sets, functions, tuples, records, and sequences on top of first-order logic.

The TLA+ toolset offers a model checker and a theorem prover: the model checker TLC enumerates states by interpreting TLA+ specifications [YML99], whereas the theorem prover TLAPS aids the user in writing and verifying interactive proofs [CDLM10]. While progress towards proof automation in TLAPS has been made in the last years [MV12a], writing interactive proofs is still a demanding task. Hence, the users prefer to run TLC for days, rather than writing proofs [NRZ15, Ong14].

In academia, several distributed algorithms were specified in TLA+: Paxos [L+01], Disk Paxos [GL03], Egalitarian Paxos [MAK13], Abstract [GKQV10], Flexible Paxos [HMS16], BFT [Lam11], and Raft [Ong14]. In industry, [NRZ15] and [Gus19] reported on finding real bugs by checking TLA+ specifications by running TLC. While TLC appears in many success stories, there is still room for improvement. Since TLC explicitly enumerates all reachable states, it suffers from the state-space explosion problem, and requires all parameters to be fixed.

In order to mitigate the state-space explosion problem, symbolic methods have been suggested in the last decades but not specific to TLA+. In these approaches, an algorithm and its desired properties are transformed to a set of logical formulas. In contrast to the explicit-state approach, a state in symbolic methods is represented implicitly as a model of such logical formulas. Then, the state space can be analyzed symbolically by using binary decision diagrams (BDDs), finite automata, or decision procedures (SAT, SMT). This saves memory since a small number of logical formulas can represent a comparatively large set of concrete states. Many symbolic tools have been developed for software/hardware verification and testing. To name a few, NuSMV [CCG+02] and LTSmin [BydPW10] implement model checking with BDDs, CBMC [KT14] and CPAChecker [BK11] implement bounded model checking [BCCZ99] and CEGAR [CGJ+03]. Domain-specific tools ByMC [KLVW17b] and Cubicle [CGK+12] prove properties of parameterized distributed...
1.1 Motivating Example 1: Two-phase Commit in TLA+

We introduce typical TLA+ constructs by discussing the famous two-phase commit protocol by [LS79] 1. In this protocol, several resource managers (e.g., databases) have to agree on whether to commit or abort a distributed transaction. The resource managers are coordinated by the transaction manager. If one of them aborts a transaction, all managers have to abort it too.

Figure 1.1 shows the TLA+ specification of two-phase commit by [GL06]. The specification is parameterized with the set of resource managers \( RM \), which, once defined, never changes in a system execution. Four variables describe the system state:

- The variable \( tmState \) stores the state of the transaction manager, which gets assigned one of the three constants “init”, “committed”, or “aborted”.
- The variable \( rmState \) is a function from a resource manager in \( RM \) to one of the four constants “working”, “prepared”, “committed”, or “aborted”.
- The variable \( tmPrepared \subseteq RM \) stores the set of resource managers that have sent a message of type “Prepared” to the transaction manager.
- The variable \( msgs \) stores the set of messages sent by the managers. It contains records of three kinds: \( [type \mapsto “Commit”] \), \( [type \mapsto “Abort”] \), and \( [type \mapsto “Prepared”, rm \mapsto S] \). The records of the third kind have an extra field \( rm \) containing a set \( S \subseteq RM \) of resource managers.

The initial system states are defined by the operator \( Init \). This operator requires \( tmState \) to be equal to “init”, the sets \( tmPrepared \) and \( msgs \) to be empty, and \( rmState \) to be a function that constrains every resource manager \( rm \in RM \) to be in the “working” state.

System transitions are defined with the operator \( Next \), which is idiomatically written as a disjunction of simpler operators, called \( actions \). In our example, there are two actions

---

1A comprehensive manual on TLA+ can be found in the book by [Lam02].
1. Introduction

---

**module** `TwoPhaseReformatted`

**constant** `RM` The set of resource managers (a parameter)

**variables**
- `rmState`, `rmState[rm]` is the state of resource manager `RM`
- `tmState`, The state of the transaction manager
- `tmPrepared`, The set of `RM` from which the `TM` has received "Prepared" messages
- `msgs`, The set of all messages sent in the distributed system

**Init\[\triangleq\]**\(\land\) `rmState = [\{rm \in RM \implies \text{"working"}\}]\) constraints on the initial states
\(\land\) `tmState = \{\text{"init"}\} \land \text{tmPrepared} = \{\}\land \text{msgs} = \{\}\)

The transitions by the transaction manager and the resource managers:

**TMRcvPrepared**\(\triangleq\) The `TM` receives a "Prepared" message from `RM` `rm`
\(\land\) `tmState = \{\text{"init"}\} \land \text{tmPrepared} = [\{\text{type} \implies \text{"Prepared"}, \text{rm} \mapsto \text{rm}\}\} \land \text{UNCHANGED} (\langle \text{rmState}, \text{tmState}, \text{msgs}\rangle)\)

The transaction manager commits the transaction:

**TMCommit**\(\triangleq\) `tmState = \{\text{"init"}\} \land \text{tmPrepared} = \text{RM} \land \text{tmState}' = \{\text{"committed"}\} \land \text{msgs}' = \text{msgs} \cup \{\{\text{type} \mapsto \text{"Commit"}\}\} \land \text{UNCHANGED} (\langle \text{rmState}, \text{tmPrepared}\rangle)\)

The transaction manager spontaneously aborts the transaction:

**TMAbort**\(\triangleq\) `tmState = \{\text{"init"}\} \land \text{tmState}' = \{\text{"aborted"}\} \land \text{msgs}' = \text{msgs} \cup \{\{\text{type} \mapsto \text{"Abort"}\}\} \land \text{UNCHANGED} (\langle \text{rmState}, \text{tmPrepared}\rangle)\)

Resource manager `rm` prepares:

**RMPrepare**\(\triangleq\) `rmState[rm] = \{\text{"working"}\} \land \text{rmState}' = [\text{rmState EXCEPT } !\{\text{rm} \mapsto \text{"prepared"}\}] \land \text{msgs}' = \text{msgs} \cup \{\{\text{type} \mapsto \text{"Prepared"}, \text{rm} \mapsto \text{rm}\}\} \land \text{UNCHANGED} (\langle \text{tmState}, \text{tmPrepared}\rangle)\)

Resource manager `rm` spontaneously decides to abort:

**RMChooseToAbort**\(\triangleq\) `rmState[rm] = \{\text{"working"}\} \land \text{rmState}' = [\text{rmState EXCEPT } !\{\text{rm} \mapsto \text{"aborted"}\}] \land \text{UNCHANGED} (\langle \text{tmState}, \text{tmPrepared}, \text{msgs}\rangle)\)

Resource manager `rm` is told by the `TM` to commit:

**RMRcvCommitMsg**\(\triangleq\) `type \mapsto \text{"Commit"}\} \in \text{msgs} \land \text{rmState}' = [\text{rmState EXCEPT } !\{\text{rm} \mapsto \text{"committed"}\}] \land \text{UNCHANGED} (\langle \text{tmState}, \text{tmPrepared}, \text{msgs}\rangle)\)

Resource manager `rm` is told by the `TM` to abort:

**RMRcvAbortMsg**\(\triangleq\) `type \mapsto \text{"Abort"}\} \in \text{msgs} \land \text{rmState}' = [\text{rmState EXCEPT } !\{\text{rm} \mapsto \text{"aborted"}\}] \land \text{UNCHANGED} (\langle \text{tmState}, \text{tmPrepared}, \text{msgs}\rangle)\)

A transition of the distributed system:

**Next\[\triangleq\]**\(\lor\) `TMCommit` \(\lor\) `TMAbort` a transition by the transaction manager
\(\lor\) `\exists rm \in RM :` a transition by the resource manager
\(\lor\) `TMRcvPrepared(rm) \lor RMPrepare(rm) \lor RMRcvCommitMsg(rm)`
\(\lor\) `RMChooseToAbort(rm) \lor RMRcvAbortMsg(rm)`

---

Figure 1.1: The two-phase commit protocol in TLA+ as specified in [GL06].
(We have only changed the indentation and comments to save some space.)
by the transaction manager and five actions by a resource manager. A resource manager is chosen with the existential quantifier $\exists rm \in RM$. The actions are TLA$^+$ formulas over two sets of variables: the variables without primes and the variables with primes. The former captures the state before a transition, while the latter captures the state after the transition.

For example, the action $RMPrepare(rm)$ is enabled when the state of $rm$ equals to “working”. This action updates the function $rmState$, so that $rmState[rm]$ becomes “prepared”, whereas the values for the other elements of $RM \setminus \{rm\}$ are not changed. Further, the action adds the record $[\text{type} \rightarrow \text{Prepared}, \text{rm} \rightarrow \text{rm}]$ to the set of messages $msgs$. Finally, the action requires that $tmState' = tmState$ and $tmPrepared' = tmPrepared$, as indicated by UNCHANGED $(tmState, tmPrepared)$.

The algorithm is designed to satisfy the following invariant:

$$\forall r_1, r_2 \in RM : rmState[r_1] \neq \text{“committed”} \lor rmState[r_2] \neq \text{“aborted”}$$ (TCConsistent)

TLA$^+$ uses syntax $f[x]$ for function application, e.g., see $rmState[rm]$. Although it looks like an array access, it is not. In contrast to arrays in programming languages, the function domains are not ordered. Hence, $f[x]$ cannot be interpreted as efficiently as an array access.

Although this example is simple in comparison to fault-tolerant protocols such as Raft [Ong14], it demonstrates several idiosyncrasies of TLA$^+$. First, there is no fixed order of evaluating the expressions. An operator such as $Next$ is just a logical formula. As soon as a vector of values for primed and non-primed variables satisfies the formula, it gives us a system transition. Second, there is no notion of an assignment. Hence, constraints on the primed variables may have different forms. Third, the language is untyped. As a result, the same variable may contain values of different types during an execution, and sets may contain type-incompatible elements.

1.2 Motivating Example 2: Chandra and Toueg Failure Detector

Our goal is to bring symbolic verification to distributed algorithms specified in TLA$^+$. To understand the application domain, we describe our motivating example the Chandra and Toueg failure detector [CT96] in this section. We also discuss challenges in verification of the failure detector.

The Chandra and Toueg failure detector can be seen as an oracle to get information about crash failures in the distributed system. The failure detector usually guarantees some of the following properties [CT96] (numbers $1..N$ denote the process identifiers):

- **Strong Accuracy**: No process is suspected before it crashes.

$$G(\forall p, q \in 1..N : (\text{Correct}(p) \land \text{Correct}(q)) \Rightarrow \neg \text{Suspected}(p, q))$$
1. Introduction

Algorithm 1.1 The eventually perfect failure detector algorithm in [CT96]

1: Every process $p \in 1..N$ executes the following:
2: for all $q \in 1..N$ do ▷ Initialization step
3: $timeout[p, q] := \text{default-value}$
4: $suspected[p, q] := \bot$
5: Send “alive” to all $q \in 1..N$ ▷ Task 1: repeat periodically
6: for all $q \in 1..N$ do ▷ Task 2: repeat periodically
7: if $suspected[p, q] = \bot$ and not hear $q$ during last $timeout[p, q]$ ticks then
8: $suspected[p, q] := \top$
9: if $suspected[p, q]$ then ▷ Task 3: when receive “alive” from $q$
10: $timeout[p, q] := timeout[p, q] + 1$
11: $suspected[p, q] := \bot$

- Eventual Strong Accuracy: There is a time after which correct processes are not suspected by any correct process.
  \[ \text{FG}(\forall p, q \in 1..N : (\text{Correct}(p) \land \text{Correct}(q)) \Rightarrow \neg \text{Suspected}(p, q)) \]
- Strong Completeness: Eventually every crashed process is permanently suspected by every correct process.
  \[ \text{FG}(\forall p, q \in 1..N : (\text{Correct}(p) \land \neg \text{Correct}(q)) \Rightarrow \text{Suspected}(p, q)) \]

where $G$ and $F$ are the globally and finally operators in linear temporal logic (LTL) respectively. Formula $G \phi$ is true for a computation path if $\phi$ holds at all states along the path. Formula $F \psi$ is true for a computation path if $\psi$ holds at some state along that path.

Given an execution trace, process $p$ is correct if $\text{Correct}(p)$ is true for every time point \(^2\). However, a process might crash later (and not recover). Given an execution trace, if process $q$ crashes at time $t$, predicate $\text{Correct}(q)$ is evaluated to false from time $t$. Predicate $\text{Suspected}(p, q)$ corresponds to the variable $suspected$ in Algorithm 1.1, and depends on the variable $timeout[p, q]$ and the waiting time of process $p$ for process $q$.

Algorithm 1.1 presents the pseudo code of the failure detector of [CT96]. A system instance has $N$ processes that communicate with each other by sending-to-all and receiving messages through unbounded $N^2$ point-to-point communication channels. A process performs local computation based on received messages (we assume that a process also receives the messages that it sends to itself). In one system step, all processes may take

\(^2\)We deviate from the original definition in [CT96], as it allows us to describe global states at specific times, without reasoning about potential crashes that happen in the future. Actually, our modeling captures more closely the failure patterns from [CT96]. Regarding the failure detector properties, the strong accuracy property is equivalent to the one in [CT96]. The other two properties have the form $FG(\ldots)$, where we may consider satisfaction only at times after the last process has crashed and thus our crashed predicates coincide with the ones in [CT96].
up to one step. Locally in each step, a process can only make a step in at most one of the locally concurrent tasks. Some processes may crash, i.e., stop operating. Correct processes follow Algorithm 1.1 to detect crashes in the system. Initially, every correct process sets a default value for a timeout of each other (Line 3), i.e., how long it should wait for others, and assumes that no processes have crashed (Line 4). Symbols $\bot$ and $\top$ refer to truth values false and true, respectively. Every correct process $p$ has three tasks: (i) repeatedly sends an “alive” message to all processes (Line 5), and (ii) repeatedly produces predictions about crashes of other processes based on timeouts (Line 6), and (iii) increases a timeout for process $q$ if $p$ has learned that its suspicion on $q$ is wrong (Line 9). Notice that process $p$ raises suspicion on the operation of process $q$ (Line 6) by considering only information related to $q$: $\text{timeout} \, [p, q]$, $\text{suspected} \, [p, q]$, and messages that $p$ has received from $q$ recently.

Algorithm 1.1 does not satisfy Eventually Strong Accuracy under asynchrony since there exists no bound on message delay, and messages sent by correct processes might always arrive after the timeout expired. Liveness of the failure detector is based on the existence of bounds $\Delta$ on the message delay, and $\Phi$ on the relative speed of processes after reaching the Global Stabilization Time (GST) at some time point $T_0$ [CT96]. There are many models of partial synchrony [DLS88 CT96]. Here we describe the case of $\Delta$ and $\Phi$ that are unknown to the processes. In this case, $T_0 = 1$, and both parameters $\Delta$ and $\Phi$ are arbitrary. Moreover, the following constraints hold in every execution:

(TC1) If message $m$ is placed in the message buffer from process $q$ to process $p$ by some operation $\text{Send}(m, p)$ at a time $s_1 \geq 1$, and if process $p$ executes an operation $\text{Receive}(p)$ at a time $s_2$ with $s_2 \geq s_1 + \Delta$, then message $m$ must be delivered to $p$ at time $s_2$ or earlier.

(TC2) In every contiguous time interval $[t, t + \Phi]$ with $t \geq 1$, every correct process must take at least one step.

These constraints make the failure detector parametric in $\Delta$ and $\Phi$.

Moreover, Algorithm 1.1 is parameterized by the number of processes and by the initial value of the timeout. If a default value of the timeout is too small, there exists a case in which sent messages are delivered after the timeout expired. This behavior violates Strong Accuracy.

As a result, verification of the failure detector faces the following challenges:

1. Its model of computation lies between synchrony and asynchrony since multiple processes can take a step in a global step.

2. The failure detector is parameterized by the number of processes. Hence, we need to verify infinitely many instances of algorithms.

---

\footnote{A time $T_0$ is called the Global Stabilization Time (GST) if $\Delta$ or $\Phi$ holds in $[T_0, \infty]$.}
3. The initial value of the timeout is an additional parameter in Algorithm 1.1.

4. The failure detector relies on a global clock and local clocks. A straightforward encoding of a clock with an integer would produce an infinite state space.

5. The algorithm is parametric by time bounds $\Delta$ and $\Phi$.

6. Eventually Strong Accuracy and Strong Completeness are liveness properties.

Moreover, if all parameters are arbitrary, we cannot check correctness of the failure detector by applying state-of-the-art formal verification techniques and tools [YML99, Ho03, CCD+14, DHV+14, DHZ16, KLVW17].

### 1.3 Problem Statements

In this thesis we focus on symbolic verification of distributed algorithms specified in TLA+. To do that, we address two following challenges:

(a) Symbolic model checking for TLA+ specifications, and

(b) Symbolic verification techniques for time constraints under partial synchrony.

To address Challenge (b), we set two sub-goals:

(c) Symbolic verification techniques for fixed time parameters, and

(d) Symbolic verification techniques for arbitrary time parameters.

### 1.4 State of the Art

In this chapter, we review the state-of-the-art on five topics: (i) general-purpose specification languages, (ii) model checkers for specialized languages, (iii) the integration of SMT solvers and interactive/semi-automatic theorem provers, (iv) cutoffs, and (v) case studies in formal verification. We divide this chapter into five sections, one for each topic.

#### 1.4.1 General-purpose Specification Languages

Specification is the process of describing a system and its desired properties [CW96]. To avoid the ambiguity and imprecision of natural languages and pseudocode, researchers have defined (formal) specification languages with a mathematical syntax and semantics.

Several general-purpose specification languages have been introduced to specify concurrent and distributed algorithms in the last decades. In addition to TLA+, an algorithm designer can choose from Alloy [Jac12], B [Abr05], IOA [GL98], VDM [Jon90], or Z [SA92]. Since these languages are widely used in both academia and industry, the conference
ABZ [KTK18] is held to compare and cross-fertilize these approaches. We briefly compare TLA+ with Alloy and B. More detailed comparisons can be found in [ABDTS18, MCI16, New14]. Table 1.1 summarizes the comparison between these languages.

Table 1.1: General-purpose specification languages and support tools

<table>
<thead>
<tr>
<th></th>
<th>TLA+</th>
<th>Alloy</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theories</td>
<td>Untyped FOL, ZFC set theory</td>
<td>Typed FOL, relational algebra</td>
<td>Typed FOL, ZF set theory</td>
</tr>
<tr>
<td>Interactive theorem prover</td>
<td>TLAPS</td>
<td>-</td>
<td>Atelier B</td>
</tr>
<tr>
<td>Automatic analysis tool and Back-end solvers</td>
<td>TLC</td>
<td>Kodkod</td>
<td>ProB</td>
</tr>
<tr>
<td></td>
<td>Explicit-state model checking</td>
<td>SAT, SMT</td>
<td>SICStus Prolog, SMT, explicit-state model checking</td>
</tr>
</tbody>
</table>

Alloy [Jac12] is a specification language combining relational algebra, first-order logic, transitive closures, integer arithmetic, and polymorphic types [Jac12]. Currently, Alloy does not support temporal operators. Therefore, it requires the user to use particular idioms to describe behavioral properties of systems. To analyse Alloy specifications, the user can run Alloy Analyzer that uses the bounded SAT-based constraint solver Kodkod [TJ07] as its back-end. Alloy and Alloy Analyzer have been used for finding bugs in distributed algorithms, e.g., in the Chord ring membership protocol [Zav12].

One shortcoming of Alloy Analyzer is that it requires the user to give precise bounds on the domain. In particular, integers have fixed bit-width, which may result in missing a counterexample. There have been a few attempts to support Alloy with SMT solvers [MRTB17, EGT11] and the tools for other languages [KSB+18, MBC+16]. The notation “-” in Table 1.1 refers to that Alloy is not currently supported by specialized theorem provers [All].

B [Abr05] is a state-based specification language rooted in predicate logic, ZF set theory, types, and arithmetic. Unlike TLA+, B requires that every member in a set has the same type. The B method has been also used in many industrial projects, e.g. railway systems [BBFM99]. Its automated analysis toolset called ProB [LB08] supports both explicit-state model checking, symbolic model checking by integrating SMT solvers [KL16, SL21], constraint solving by using SICStus Prolog [CWA+88]. Several tools translate fragments of B to other specification languages, e.g. to TLA+ [HL12, PL12] and Alloy [KSB+18]. To verify a B specification, the user can use the interactive theorem prover Atelier B [Cle]. Chapters 3 and 4 are related to the translation by [HL12] from TLA+ to B. This translation allows one to apply the model checker ProB, which uses constraint solving. As B and TLA+ are much closer than SMT and TLA+, their translation is conceptually simpler, though the authors had to deal with a few language incompatibilities. The authors of [HL12] report on the experiments with SimpAlloc, as well as with simpler benchmarks, where ProB was shown to be more efficient than TLC. However, we are not aware of applying this tool to fault-tolerant distributed algorithms.
such as Paxos or the two-phase commit protocol. The authors of [SL21] provide a translation from B to SMT-LIB that utilizes lambda expressions supported in Z3 to encode operators in B, e.g., set comprehension and set operators.

1.4.2 Model Checkers for Specialized Languages

Promela is the input language of the model checker Spin [Hol03]. Promela supports Boolean and integer variables, arrays, processes, message channels, arithmetic, and temporal operators. Spin is an explicit-state model checker that was applied to many industrial problems. Moreover, the authors of [DTT14] checked a version of Paxos, and the authors of [Zav15] checked Chord. While we could encode the benchmarks in Promela, this work requires serious efforts, as specifications in Promela are low-level in comparison to TLA$^+$. NuSMV [CCG02] and nuXMV [CCD14] stem from the symbolic model checker SMV [McM93]. They are designed for modeling finite-state hardware protocols. The SMV language is much more restrictive than TLA$^+$. This makes it very hard to use for verification of distributed algorithms.

Several techniques and tools for parameterized verification of fault-tolerant distributed algorithms were introduced by [MTK08, MTK09, DHV14, DHZ16, FKP16, vGBR16, KLVW17b, MSB17]. The efficiency of these techniques comes from the restriction to special domains, whereas our approach applies to virtually any TLA$^+$ specification over finite structures.

ByMC (Byzantine Model Checker) [KW18] implements verification methods for threshold-guarded distributed algorithms. Such algorithms have the following features: (1) at most $T$ of processes may crash or be Byzantine; (2) the number $N$ of processes in the system is a parameter, as well as $T$; (3) correct processes count messages and progress when they receive sufficiently many messages, e.g., at least $T + 1$; (4) and the parameters are restricted by a resilience condition, e.g., $3T < N$. ByMC was used to verify many threshold-guarded distributed algorithms [KW18, KLSW20].

Finally, symbolic model checking has been applied in many different application domains. For example, TAMARIN [MSCB13] focuses on security protocols, and Kind 2 [CMST16] is designed for the dataflow language Lustre.

1.4.3 Integration of SMT Solvers and Theorem Provers

In this section, we first provide an overview of the recent works on integrating SMT solvers into interactive theorem provers. Then, we describe semi-automatic theorem provers whose back-end engines are SMT solvers.

Interactive Theorem Provers and SMT Solvers. [MV18] introduced two encodings to translate TLA$^+$ to SMT formulas: an untyped one and a multi-sorted one. Their work is designed towards proving unsatisfiability of obligations inside the TLA Proof
1.4. State of the Art

System [CDLM10]. These obligations are typically small in comparison to a complete TLA\(^+\) specification, and their techniques utilize quantified formulas which are supported by SMT fairly well for the unsatisfiable case. If SMT solvers cannot decide on satisfiability, the user has to prove the obligation manually.

Sledgehammer is a tool to combine the interactive theorem prover Isabelle [NPW02] with a variety of automatic theorem provers (ATPs) and SMT solvers [PS07, BBP13]. Since Isabelle is designed for polymorphic higher-order logic, the translation meets challenges in higher-order features and type information. Moreover, Sledgehammer’s success rate depends on lemmas extracted from Isabelle’s libraries by a relevance filter, and on heuristics to instantiate quantifiers, e.g. weights and triggers.

SMTCooq [EMT+17] is a plug-in for integrating SMT and SAT solvers into the interactive theorem prover Coq [BC13]. The primary use case for SMTCooq aims at increasing the level of automation in Coq. SMTCooq provides tactics to translate a Coq goal into SMT expressions that use uninterpreted functions, linear integer arithmetic, bit vectors, and functional arrays. When the SMT solver produces a proof certificate, SMTCooq validates the certificate and generates a Coq proof for the original goal.

Semi-Automated Provers using Decision Procedures. Ivy is a multi-modal verification tool for correct design and implementation of distributed protocols and algorithms [MP20]. The main characteristic of Ivy is that its language is designed with decidable theories and reasoning in mind. For example, the effectively propositional fragment (EPR) and stratified function symbols that are decidable fragments in first-order logic are currently supported in Ivy (version 1.7) [McM]. This feature makes Ivy stable and predictable in reasoning about specifications with decidable problems. When a decidable specification has a bug, Ivy is usually able to provide a counterexample. If a specification is big and complicated, the user can decompose it into smaller component modules and provide proof hints to Ivy. Researchers have proved safety of several variants of Paxos [PLSS17] and Tendermint consensus [Gal] with Ivy.

[BCD+05, Lei08] developed the intermediate verification language Boogie, which serves as a layer on which to build program verifiers for other languages, e.g. VCC [CDH+09], Dafny [Lei10], and Spec# [BLS04]. Boogie expressions are translated to the input languages of automatic theorem provers, primarily to the SMT solver Z3, by applying Hoare logic [Hoa69]. This approach brings a higher degree of automation, but does not eliminate the human proof effort required since Boogie uses undecidable theories of SMT. The main application of Dafny is verification of sequential programs, whereas TLA\(^+\) is built around non-determinism.

[SHK+16] designed the general-purpose functional programming language F* with effects aimed at program verification. Like Boogie, this language utilizes SMT solvers as back-end provers, and supports interactive proofs. This language targets to fill in the gap between implementation and verification.
1. Introduction

1.4.4 Cutoffs

Distributed algorithms are typically parameterized in the number of participants, e.g., two-phase commit protocol in Section 1.1 and the Chandra and Toueg failure detector in Section 1.2. While the general parameterized verification problem is undecidable [AK86, Suz88, BJK+15], many distributed algorithms such as mutual exclusion and cache coherence enjoy the cutoff property, which reduces the parameterized verification problem to verification of a small number of instances. In a nutshell, a cutoff for a parameterized algorithm \( A \) and a property \( \phi \) is a number \( k \) such that \( \phi \) holds for every instance of \( A \) if and only if \( \phi \) holds for instances with \( k \) processes [EN95, BJK+15]. In the last decades, researchers have proved the cutoff results for various models of computation: ring-based message-passing systems [EN95, EK04], purely disjunctive guards and conjunctive guards [EK00, EK03], token-based communication [CTTV04], and quorum-based algorithms [MSB17]. However, we cannot apply these results to the failure detector in Section 1.2 because it relies on point-to-point communication and timeouts. Moreover, distributed algorithms discussed in [EN95, EK00, EK03, EK04, CTTV04, MSB17] are not in the symmetric point-to-point class.

1.4.5 Formal Verification for Partial Synchrony Constraints

Partial synchrony is a well-known model of computation in distributed computing. To guarantee liveness properties, many practical protocols, e.g., the failure detector in Section 1.2 and proof-of-stake blockchains [BKM18, YMR+19], assume time constraints under partial synchrony. That is the existence of bounds \( \Delta \) on message delay and \( \Phi \) on the relative speed of processes after some time point.

While partial synchrony is important for system designers, it is challenging for verification. The mentioned constraint makes partially synchronous algorithms parametric in time bounds. Moreover, partially synchronous algorithms are typically parameterized in the number of processes.

Research papers about partially synchronous algorithms, including papers about failure detectors [LAF99, ADGFT06, ADGFT08] contain manual proofs and no formal specifications. Without these details, proving those distributed algorithms with interactive theorem provers [CDL+12, MP20] is impossible.

Data-independence techniques have been used to verify a class of distributed protocols parameterized by their network topology [CR99]. A protocol is data-independent with respect to a data type when the only operation it can perform on values of that type is equality testing [Wol86, LR99]. However, the Chandra and Toueg failure detector in Section 1.2 requires the addition operation on natural numbers.

System designers can use timed automata [AD94] and parametric verification frameworks [LPY97, AFKS12, LRST09] to specify and verify timed systems. In the context of timed systems, we are aware of only one paper about verification of failure detectors [AMOI2]. In this paper, the authors used three tools, namely UPPAAL [LPY97]...
mCRL2 \cite{BGK+19}, and FDR2 \cite{Ros10} to verify small instances of a failure detector based on a logical ring arrangement of processes. Their verification approach required that message buffers were bounded, and had restricted behaviors in the specifications. Moreover, they did not consider the bound $\Phi$ on the relative speed of processes. In contrast, there are no restrictions on message buffers, and no ring topology in the Chandra and Toueg failure detector in Section 1.2.

In recent years, automatic parameterized verification techniques \cite{KLVW17a,KLVW17b,SKWZ19,DWZ20} have been introduced for distributed systems, but they are designed for synchronous and/or asynchronous models. Interactive theorem provers have been used to prove correctness of distributed algorithms recently. For example, researchers proved safety of Tendermint consensus with IVy \cite{Gal}.

### 1.5 Methodology

Our goal is to bring symbolic verification to distributed algorithms under partial synchrony. To do that, we first need to specify distributed algorithms in a formal specification language. In this thesis, we focus on specifications written in TLA$^+$, which has been applied to formalize many distributed algorithms.

#### 1.5.1 Symbolic Model Checking for TLA$^+$

Many symbolic model checking techniques and tools for hardware and/or software verification have been suggested in the last decades. However, these techniques are not specific to TLA$^+$. Hence, we first focus on general-purpose symbolic model checking for TLA$^+$ (Challenge (a)). In this step, we assume that every parameter is fixed.

Symbolic reasoning is typically based on binary decision diagrams (BDDs) \cite{Bry86}, finite automata \cite{RMM+97}, or decision procedures (SAT, SMT) \cite{BCCZ99,CHVB18}. Since many distributed algorithms contain integers, functions, and sets, first-order logic allows natural formulations of these data structures. So, we use first-order logic as a symbolic representation, and run SMT solvers to find models of logical constraints.

Because TLA$^+$ and SMT have different levels of expressiveness, we design a reduction system from TLA$^+$ to SMT. To make the tool usable for the TLA$^+$ community, we aim at introducing as few restrictions to the language as the explicit-state model checker TLC does. Hence, whenever we have a choice between an efficient SMT encoding that restricts the input and a less efficient but general SMT encoding, we choose the general one.

System transitions in TLA$^+$ are typically defined in a single logical formula. For example, see formula $\text{Next}$ in Figure 1.1. Instead of directly reducing $\text{Next}$ to a SMT formula, we would like to slice $\text{Next}$ into symbolic transitions, and then reduce them into SMT. The main challenge in finding symbolic transitions in a TLA$^+$ specification is the lack of control flow statements, e.g., assignments.

13
1. Introduction

Currently, TLC interprets an equality $x' = e$ as an assignment of $e$ to the yet unbound variable $x$. Inspired by TLC, we develop an automatic technique for discovering expressions in TLA$^+$ formulas such as $x' = e$ and $x' \in \{e_1, \ldots, e_k\}$ that can be provably used as assignments. In contrast to TLC, our technique does not explicitly evaluate expressions, but it reduces the problem of finding assignments to the satisfiability of an SMT formula. This technique is implemented in a preprocessing step of our model checker APALACHE.

Figure 1.2 shows the main phases of APALACHE. The dashed area shows the preprocessing phases.

1.5.2 Symbolic Verification Techniques for Partial Synchrony Constraints

After having a symbolic model checker for TLA$^+$, we focus on verification methods to cope with partial synchrony constraints.

In addition to time bounds $\Delta$ and $\Phi$, partial synchrony algorithms might have other prerequisites. We find that the failure detector [CT96] is an interesting case. While the failure detector is a basic building block of fault-tolerant distributed algorithms, its correctness is based only on the existence of time bounds $\Delta$ and $\Phi$ from the Global Stabilization Time (GST). To make a decision about the status of process $q$, process $p$ needs to consider only the existence of received messages from process $q$. Hence, we choose the failure detector as a case study in our thesis (Challenge (b)).

We can apply our symbolic model checker to analyze the failure detector in case of fixed parameters. However, the failure detector is (i) parameterized in the number of processes and a default value of the timeout, and (ii) parametric in time bounds. Therefore, this calls for further verification methods to verify all of its instances.

Assuming that the time bounds are fixed, we still have to verify infinitely-many instances since the number of processes in the failure detector is arbitrary (Challenge (c)). To cope with an arbitrary number of participants in a distributed algorithm, a well-known approach is to prove cutoff results for this algorithm [EN95, CTTV04]. In a nutshell, given a property $\xi$ and a system that has a parameter $m$, a cutoff is a number $B \geq 1$ such that whenever all instances that assign a value not greater than $B$ to a parameter $m$ satisfy $\xi$, then all instances which assign an arbitrary number to $m$ satisfy $\xi$. Hence, verification of algorithms that enjoy the cutoff property can be done by model checking of finite instances. In this thesis, we prove cutoff results of the failure detector. Moreover,
we define a new class of distributed algorithms that includes the failure detector, and
generalize these cutoff results for every algorithm in this class.

To verify the failure detector in case of arbitrary time bounds (Challenge(d)), we write
a mechanized proof in the interactive theorem prover IVy [MP20]. A typical way to
prove correctness of distributed algorithms with a proof assistant is based on inductive
invariants. We first find and check inductive invariants of the failure detector in case of
fixed parameters with TLC and our symbolic model checker APALACHE. Then, we
construct inductive invariants for cases of arbitrary time bounds, and check them with
IVy. Moreover, to reduce complexity of the proofs, we develop encoding techniques to
efficiently specify the failure detector, and an abstraction on in-transit messages. These
methods allow us to reduce complexity of process behavior, and the number of in-transit
messages.

1.5.3 Case Studies and Applications in Formal Verification

Case study research is one of several methodologies employed for empirical studies of
formal verification. A case study is typically defined as “an empirical inquiry that
investigates a contemporary phenomenon within its real-life context” [R+03]. Through
case studies, researchers test existing hypotheses, generate new hypotheses, have a deeper
understanding of why something happened, and know what might be important to
investigate in future research [R+03, Fly06].

Many formal languages and support tools have been introduced in the last decades. To
compare the strengths and weaknesses of verification systems, case studies have been
carried out. For example, Zav15 made a comparison between Alloy and Spin. New14
presented their evaluation of Alloy, Microsoft VCC [CDH+09], and TLA+ by trying
these tools on real-world problems.

In addition, case studies have been used to demonstrate that verification systems have
reached a level of maturity that allows us to tackle interesting problems. For instance,
PQ09 presented how to verify controllability, reactivity, safety, and liveness properties
of European Train Control System with KeYmaera. Ler09 reported on the development
and formal verification of the compiler CompCert with the use of Coq in programming
the compiler and in proving its correctness. New14 reported that TLC found bugs in
Amazon distributed systems. HHK+15 demonstrated the IronFleet methodology on
an implementation of a Paxos-based replicated state machine library and a lease-based
sharded key-value store, and on proof of their correctness.

Case studies in formal verification usually present challenges that the user faces when
applying verification tools to analyze a problem, together with corresponding solutions,
e.g., see Ler09, PQ09, New14. Moreover, such case studies usually come up with
recommendations on the future development of verification frameworks.

Our model checker APALACHE has been significantly updated since it was introduced.
In Chapter 5, we evaluate our model checker APALACHE version 0.16.2 by conducting
a case study on model checking of the Skeen atomic multicast algorithm Ske85.
1. Introduction

A shortcoming of APALACHE is that every parameter in a given specification must be fixed. Therefore, we need additional techniques to verify a partial synchronous algorithm, which has arbitrary time bounds. In Chapter 7, we demonstrate our approach to verify distributed algorithms under partial synchrony by conducting a case study on verification of the Chandra and Toueg failure detector in Section 1.2 by applying different symbolic verification tools. In this work, we first to verify inductive invariants of small instances with APALACHE. Then, we generalize them for cases of arbitrary bounds and verify the results with the theorem prover IVy.

1.6 Publications

This dissertation contains text and material from the following conference papers.


Three publications [KTK18], [KKT19], and [TKW21b] are to address Challenge (a). The results in [KTK18] are used as a preprocessing step in [KKT19]. The latest version of APALACHE is evaluated in [TKW21b]. Two papers [TKW20] and [TKW21a] are to address Challenge (b). More specifically, [TKW20] is to address Challenge (c), and [TKW21a] is to address Challenge (d).

Furthermore, the following journal articles are extended results of the mentioned papers.


The following describes the contribution of the thesis author in the mentioned conference papers. The thesis author is the main author in three papers [TKW20], [TKW21a]
Igor Konnov and Josef Widder are Ph.D. supervisors. The thesis author is one of two main authors in [KTK18] and [KKT19].

- In [KTK18], the thesis author proved soundness of the method to extract symbolic transitions from TLA+ specifications, and prepared the benchmarks.
- In [KKT19], the thesis author proved soundness of the reduction system to translate TLA+ expressions to quantifier-free SMT constraints, and conducted experimental evaluation.

1.7 Structure of the Thesis

The remainder of this dissertation is structured into the two main parts.

Part [I] focuses on APALACHE – a general-purpose symbolic model checker for TLA+. In Chapter 2, we describe how to extract symbolic transitions from TLA+ specifications. We present APALACHE in Chapter 3. We prove soundness of our reduction system from TLA+ to SMT in Chapter 4. We present a case study on model checking of the Skeena atomic multicast algorithm with TLC and APALACHE in Section 5.

Part [II] focuses on symbolic verification of symmetric point-to-point distributed algorithms under partial synchrony. We define the class of symmetric point–to–point distributed algorithms and prove cutoff results on the number of processes in Chapter 6. This class contains our motivating example, the Chandra and Toueg failure detector. We provide a case study on verification of the Chandra and Toueg failure detector under partial synchrony by using the tools: TLA+ and its model checker, counter automata, and the interactive theorem prover IVy in Chapter 7. In this case study, our verification results are for cases of unknown time bounds and an arbitrary default value of the timeout.
Part I

A General-purpose Symbolic Model Checker for TLA$^+$
CHAPTER 2

Extracting Symbolic Transitions from TLA\(^+\) specifications

In TLA\(^+\), a system specification is written as a logical formula that restricts the system behavior. As a logic, TLA\(^+\) does not have assignments and other imperative statements that are used by model checkers to compute the successor states of a system state. Model checkers compute successors either explicitly — by evaluating program statements — or symbolically — by translating program statements to an SMT formula and checking its satisfiability. To efficiently enumerate the successors, TLA’s model checker TLC introduces side effects. For instance, an equality \(x = e\) is interpreted as an assignment of \(e\) to the yet unbound variable \(x\).

Inspired by TLC, we introduce an automatic technique for discovering expressions in TLA\(^+\) formulas such as \(x' = e\) and \(x' \in \{e_1, \ldots, e_k\}\) that can be provably used as assignments. In contrast to TLC, our technique does not explicitly evaluate expressions, but it reduces the problem of finding assignments to the satisfiability of an SMT formula. Hence, we give a way to slice a TLA\(^+\) formula in symbolic transitions, which can be used as an input to a symbolic model checker. Our prototype implementation successfully extracts symbolic transitions from several TLA\(^+\) benchmarks.

This chapter presents an extended version of the paper at ABZ’18 [KTK18] and partially of the journal paper at SCP’20 [KTK20].

2.1 Overview

We start this section by discussing our motivating example. A simple example in Figure 2.1 illustrates a problem that one faces when developing a symbolic model checker for TLA\(^+\). In this example, we model two processes: \textit{Producer} that inserts a subset of \{“A”, “B”, “Z”, “1”, “8”\} into the set \(S\), and \textit{Consumer} that removes from \(S\) its arbitrary
2. Extracting Symbolic Transitions from TLA+ specifications

---

**MODULE prodcons**

VARIABLE $S$, empty

\[ Init \triangleq S = \{\} \land empty = \text{TRUE} \]

\[ Produce \triangleq \land empty' = \text{FALSE} \land \exists X \in \text{subset} \{“A”, “B”, “Z”, “1”, “8”\} : S' = S \cup \{X\} \]

\[ Consume \triangleq \neg empty \land S' \in \text{subset} S \land empty' = (S' = \{\}) \]

\[ Next \triangleq \text{Produce} \lor \text{Consume} \]

---

Figure 2.1: A simple producer-consumer subset. The system is initialized with the operator \textit{Init}. A system transition is specified with the operator \textit{Next} that is defined via a disjunction of operators \textit{Produce} and \textit{Consume}. Both Producer and Consumer maintain the state invariant \textit{empty} $\equiv (S = \emptyset)$. We notice the following challenge for a symbolic approach: Direct translation of \textit{Next} to SMT would produce a monolithic formula, e.g., it would not analyze \textit{Produce} and \textit{Consume} as independent actions. This is in sharp contrast to translation of imperative programs, in which variable assignments allow a model checker to focus only on the local state changes.

Our motivation comes from the observation on how TLC computes the successors of a given state [Lam02, Ch. 14]. Instead of precomputing all potential successors and evaluating \textit{Next} on them, TLC explores subformulas of \textit{Next}. The essential exploration rules are: (1) Disjunctions and conjunctions are evaluated from left to right, (2) an equality $x' = e$ assigns the value of $e$ to $x'$ if $x'$ is yet unbound, (3) if an unbound variable appears on the right-hand side of an assignment or in a non-assignment expression, TLC terminates with an error, and (4) operands of a disjunction assign values to the variables independently. In more detail, rule (4) means that whenever a disjunction $A \lor B$ is evaluated and $x'$ is assigned a value in $A$, this value does not propagate to $B$; moreover, $x'$ must be assigned a value in $B$.

In our example, TLC evaluates the actions \textit{Produce} and \textit{Consume} independently and assigns variables as prescribed by these formulas. As TLC is explicit, for each state, it produces at most $2^2$ successors in \textit{Produce} as well as in \textit{Consume}.

We introduce a technique to statically label expressions in a TLA+ formula $\phi$ as assignments to the variables from a set $V'$, while fulfilling the following:

1. For purely Boolean formulas, if one transforms $\phi$ into an equivalent formula $\lor_{1 \leq i \leq k} D_i$ in disjunctive normal form (DNF), then every disjunct $D_i$ has exactly one assignment per variable from $V'$.

2. The assignments adhere to the following partial order: if $x' \in V'$ is assigned a value in expression $e$, that uses a variable $y' \in V'$, then the assignment to $y'$ precedes the assignment to $x'$.
2.2 Abstract Syntax $\alpha$-TLA$^+$

In general, we formalize the above idea with the notion of a branch.

As expected, the following sequence of expressions is given as assignments in our example:

1. $empty' = true$
2. $S' = S \cup \{X\}$
3. $S' \in \text{subset } S$
4. $empty' = (S' = \emptyset)$

Using this sequence, our technique constructs two symbolic transitions that are equivalent to the actions $Produce$ and $Consume$.

In general, finding assignments and slicing a formula into symbolic transitions is not as easy as in our example, because of quantifiers and if-then-else complicating matters. In this chapter, we present our solution, demonstrate its soundness and report on preliminary experiments.

**Structure.** This chapter is organized as follows: Section 2.2 introduces an abstraction of TLA$^+$ syntax, called $\alpha$-TLA$^+$, which preserves only those language constructs, that are useful for determining assignments. In Section 2.3 we introduce auxiliary notions, such as label sets, assignment candidates, and the dependency relation. Section 2.4 introduces branches – Boolean formulas abstracting the structure of $\alpha$-TLA$^+$ – and the definition of an assignment strategy, in terms of its branch properties. Section 2.5 presents the encoding of assignment strategies into SMT. In Section 2.6 we use the results of the previous section to recover information about the original TLA$^+$ formula; we introduce the notion of slices and a specific subset thereof, symbolic transitions. Finally, Section 2.7 details experimental results.

2.2 Abstract Syntax $\alpha$-TLA$^+$

To focus only on the expressions that are essential for finding assignments in a formula, we define abstract syntax for TLA$^+$ formulas below. In our syntax, the essential operators such as conjunctions and disjunctions are included explicitly, while the other non-essential operators are hidden under the star expression $\ast$.

We assume predefined three infinite sets:

- A set $\mathcal{L}$ of labels. We use notation $\ell_i$ to refer to its elements, for $i \in \mathbb{N}$.
- A set $\text{Vars}'$ of primed variables that are decorated with prime, e.g., $x'$ and $a'$.
- A set $\text{Bound}$ of bound variables, which are used by quantifiers.

The abstract syntax $\alpha$-TLA$^+$ is defined in terms of the following grammar:

$$
\text{expr} ::= \text{ex}_\alpha \mid \ell ::= \mathbf{F} \\
| \ell ::= v' \in \text{ex}_\alpha \mid \ell ::= \text{expr} \land \cdots \land \text{expr} \mid \ell ::= \text{expr} \lor \cdots \lor \text{expr} \\
| \ell ::= \exists x \in \text{ex}_\alpha : \text{expr} \mid \ell ::= \text{ite}(\text{ex}_\alpha, \text{expr}, \text{expr}) \\
\text{ex}_\alpha ::= \ell ::= \ast (v', \ldots, v') \\
\ell ::= \text{a unique label from the set } \mathcal{L} \\
v' ::= \text{a variable name from the set } \text{Vars}'
$$
Next $\triangleq \ell_1 :: (\ell_2 :: (\ell_3 :: \text{empty}' \in \ell_4 :: \ast \land \ell_5 :: \exists X \in \ell_6 :: \ast : \ell_7 :: S' \in \ell_8 :: \ast) \\
\lor \ell_9 :: (\ell_{10} :: \ast \land \ell_{11} :: S' \in \ell_{12} :: \ast \land \ell_{13} :: \text{empty}' \in \ell_{14} :: \ast(S'))$

Figure 2.2: The Next operator of producer-consumer in $\alpha$-TLA$^+$

$x ::= \text{a variable name from the set } Bound$

A few comments on the syntax and its relation to TLA$^+$ expressions are in order. We require every expression to carry a unique label $\ell_i \in L$. Although this is not a requirement in TLA$^+$, it is easy to decorate every expression with a unique label. The expressions of the form $\ell :: v' \in expr$ are of ultimate interest to us, as they are treated as assignment candidates. Under certain conditions, they can be used to assign to $v'$ a value from the set represented by the expression $expr$. Perhaps somewhat unexpectedly, expressions such as $v' = e$ and UNCHANGED $(v_1, \ldots, v_k)$ are not included in our syntax. To keep the syntax minimal, we represent them with $\ell :: v' \in expr$. Indeed, these expressions can be rewritten in an equivalent form: $v' = e$ as $v' \in \{e\}$, and UNCHANGED $(v_1, \ldots, v_k)$ as $v'_1 \in \{v_1\} \land \cdots \land v'_k \in \{v_k\}$.

Every non-essential TLA$^+$ expression $e$ is presented in the abstract form $\ell :: \ast(v'_1, \ldots, v'_k)$, where $v'_1, \ldots, v'_k$ are the names of the primed variables that appear in $e$. When no primed variable appears in an expression, we omit parenthesis and write $\ell :: \ast$. TLA$^+$ expressions often refer to user-defined operators, which are not present in our abstract syntax. We simply assume that all non-recursive user-defined operators have been expanded, that is, recursively replaced with their bodies. All uses of recursive operators are hidden under $\ast$; hence, recursive operator definitions are ignored when searching for assignment candidates.

It should be now straightforward to see how one could translate a TLA$^+$ expression to our abstract syntax. We write $\alpha(e)$ to denote the expression in $\alpha$-TLA$^+$ that represents an expression $e$ in the complete TLA$^+$ syntax. With $\gamma$ we denote the reverse translation from $\alpha$-TLA$^+$ to TLA$^+$ that has the property that $\gamma(\alpha(e)) = e$. Figure 2.2 shows the abstract expression $\alpha(\text{Next})$ of the operator Next defined in Figure 2.1.

Discussions. Notice that $\alpha$-TLA$^+$ is missing several fundamental constructs permitted in TLA$^+$, such as CASE expressions, universal quantifiers, and negations. They are all abstracted to $\ast$. The primary purpose of $\alpha$-TLA$^+$ is to allow us to determine whether a given expression containing set inclusion — or equality — can be used as an assignment. If such an expression occurs under a universal quantifier, it is not clear which value should be used for an assignment. Hence, we abstract the expressions under universal quantifiers. For a similar reason, we abstract the expressions under negation. The latter is consistent with TLC, which produces an error when given, for example, $\text{Next} == \neg(x' = 1)$. Finally, we abstract CASE, due to its semantics, which is defined in terms of the CHOOSE operator [Lam02, Ch. 6]. In practice, there are no potential assignments under CASE in the standard TLA$^+$ examples.
2.3. Preliminary Definitions

Every TLA+ specification declares a certain finite set of variables, which may appear in the formulas contained therein. Let \( \phi \) be an \( \alpha \)-TLA+ expression. We assume, for the purposes of our analysis, that \( \phi \) is associated with some finite set \( \text{Vars}'(\phi) \), which is a subset of \( \text{Vars}' \), containing all of the variables that appear in \( \phi \) (and possibly additional ones). This is the set of variables declared by the specification in which \( \gamma(\phi) \) appears.

Since the labels are unique, we write \( \text{lab}(\ell : \psi) \) to refer to the expression label \( \ell \) and \( \text{expr}(\ell) \) to refer to the expression that is labeled with \( \ell \). We refer to the set of all subexpressions of \( \phi \) by \( \text{Sub}(\phi) \). See Table 2.1 for a formal definition.

<table>
<thead>
<tr>
<th>( \alpha )-TLA+ expression ( \phi )</th>
<th>( \text{Sub}(\phi) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \ell : * (v'_1, \ldots, v'_k) )</td>
<td>( { \phi } )</td>
</tr>
<tr>
<td>( \ell : v' \in \phi_1 )</td>
<td>( { \phi, \phi_1 } )</td>
</tr>
<tr>
<td>( \ell : \bigwedge_{i=1}^s \phi_i ) or ( \ell : \bigvee_{s=1}^i \phi_i )</td>
<td>( { \phi } \cup \bigcup_{i=1}^s \text{Sub}(\phi_i) )</td>
</tr>
<tr>
<td>( \ell : \exists x \in \phi_1 : v_2 )</td>
<td>( { \phi } \cup \text{Sub}(\phi_1) \cup \text{Sub}(\phi_2) )</td>
</tr>
</tbody>
</table>

Table 2.1: The definition of \( \text{Sub}(\phi) \)

The set \( \text{Sub}(\phi) \) allows us to reason about terms that appear inside an expression \( \phi \), at some unknown/irrelevant depth. We will often refer to the set of all labels appearing in \( \phi \), that is, \( \text{Labs}(\phi) = \{ \text{lab}(\psi) \mid \psi \in \text{Sub}(\phi) \} \).

Of special interest to us are assignment candidates, i.e., expressions of the form \( \ell : v' \in \phi_1 \). Given a variable \( v' \in \text{Vars}'(\phi) \) and an \( \alpha \)-TLA+ expression \( \phi \), we write \( \text{cand}(v', \phi) \) to mean the set of labels that belong to assignment candidates for \( v' \) in subexpressions of \( \phi \). More formally, \( \text{cand}(v', \phi) = \{ \ell \mid (\ell : v' \in \psi) \in \text{Sub}(\phi) \} \). An exhaustive definition is included in [KTK20]. We use the notation \( \text{cand}(\phi) \) to mean \( \bigcup_{v' \in \text{Vars}'(\phi)} \text{cand}(v', \phi) \).

Finally, we assign to each label \( \ell \in \text{Labs}(\phi) \) a set \( \text{frozen}_\phi(\ell) \subseteq \text{Vars}'(\phi) \). Intuitively, if a variable \( v' \) is in \( \text{frozen}_\phi(\ell) \), then no expression of the form \( \hat{\ell} : v' \in \psi \) can be treated as an assignment inside \( \text{expr}(\ell) \). Formally, for every \( \ell \in \text{Labs}(\phi) \) the set \( \text{frozen}_\phi(\ell) \) is defined as the minimal set satisfying all the constraints in Table 2.2.

The sets \( \text{frozen}_\phi \) naturally lead to the dependency relations \( \preceq_{v'} \) on \( \text{Labs}(\phi) \), where \( v' \in \text{Vars}'(\phi) \). We will use \( \ell_1 \prec_{v'} \ell_2 \) to mean that \( \ell_1 \) is an assignment candidate for \( v' \),
which also belongs to the frozen set of $\ell_2$. Formally:

$$\ell_1 \triangleleft_{v'} \ell_2 \iff \ell_1 \in \text{cand}(v', \phi) \land v' \in \text{frozen}_\phi(\ell_2)$$

Intuitively, if $\ell_1 \triangleleft_{v'} \ell_2$ we want to make sure that $\text{expr}(\ell_1)$ is evaluated before $\text{expr}(\ell_2)$, if possible.

**Remark 2.3.1.** Let us look at the following $\alpha$-TLA$^+$ expression:

$$\ell_1 :: [\exists i \in [\ell_2 :: * (y')]: \ell_3 :: x' \in [\ell_4 :: *]]$$

Take the subexpression $\ell_3 :: x' \in [\ell_4 :: *]$, which we name $\psi$. By solving the constraints for $\text{frozen}_\phi(\ell_3)$ we conclude that $\text{frozen}_\phi(\ell_3) = \emptyset$. However, if we take the additional constraints for $\text{frozen}_\phi(\ell_3)$ into consideration, the empty set no longer satisfies all of them, specifically, it does not satisfy the condition imposed by the existential quantifier in $\ell_1$. The additional requirement $\{y'\} \subseteq \text{frozen}_\phi(\ell_3)$ implies that $\text{frozen}_\phi(\ell_3) = \{y'\}$. This corresponds to the intuition that expressions under a quantifier, like $\psi$, implicitly depend on the bound variable and the expressions used to define it, which is $\text{expr}(\ell_2)$ in our example.

### 2.4 Formalizing Symbolic Assignments

As TLC evaluates formulas in a left-to-right order, there is a very clear notion of an assignment; the first occurrence of an expression $v' \in S$ is interpreted as an assignment to $v'$. In our work, we want to *statically* find expressions that can safely be used as assignments. If we were only dealing with Boolean formulas, we could transform the original TLA$^+$ formula to DNF, $\bigvee_{i=1}^s D_i$, and treat each $D_i$ independently. However, we also need to find assignments, which may be nested under existential quantifiers. To transfer our intuition about DNF to the general case we first introduce a transformation boolForm that captures the Boolean structure of the formula. Then, we introduce branches and assignment strategies to formalize the notion of assignments in the symbolic case.

**Boolean structure of a formula and branches.** The transformation boolForm maps an $\alpha$-TLA$^+$ expression to a Boolean formula over variables from $\{b_{\ell} \mid \ell \in \mathcal{L}\}$. The definition of boolForm can be found in Table 2.3. As boolForm($\phi$) is a formula in Boolean logic, a model of boolForm($\phi$) is a mapping from $\{b_{\ell} \mid \ell \in \mathcal{L}\}$ to $\mathbb{B} = \{\text{true}, \text{false}\}$.

Take $S \subseteq \mathcal{L}$. The set $S$ naturally defines a model induced by $S$, denoted $\mathcal{M}[S]$, by the requirement that $\mathcal{M}[S] \models b_{\ell}$ if and only if $\ell \in S$.

The boolForm transformation allows us to formulate the central notion of a branch: A set $Br \subseteq \mathcal{L}$ is called a *branch* of $\phi$ if the following constraints hold:

(a) The set $Br$ induces a model of boolForm($\phi$), i.e., $\mathcal{M}[Br] \models \text{boolForm}(\phi)$, and
2.4. Formalizing Symbolic Assignments

Table 2.3: The definition of boolForm(φ)

<table>
<thead>
<tr>
<th>α-TLA+ expression φ</th>
<th>boolForm(φ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ℓ :: F or ℓ :: *(v'_1, ..., v'_k) or ℓ :: v' ∈ φ_1</td>
<td>b_ℓ</td>
</tr>
<tr>
<td>ℓ :: ∨_i=1 φ_i</td>
<td>∨_i=1 boolForm(φ_i)</td>
</tr>
<tr>
<td>ℓ :: ∃x ∈ φ_1: φ_2</td>
<td>boolForm(φ_2)</td>
</tr>
<tr>
<td>ℓ :: ITE(φ_1, φ_2, φ_3)</td>
<td>boolForm(φ_2) ∨ boolForm(φ_3)</td>
</tr>
</tbody>
</table>

(b) The model M[Br] is minimal, that is, M[S] ≠ boolForm(φ) for every S ⊂ Br.

Then, Branches(φ) is the set of all branches of φ.

Remark 2.4.1. Let us look the α-TLA+ expression φ given by

\[ ℓ_1 :: [[ℓ_2 :: x' ∈ *] ∧ [ℓ_3 :: [[ℓ_4 :: x' ∈ *] ∨ [ℓ_5 :: x' ∈ *]]]] \]

We know that boolForm(φ) = b_ℓ_1 ∧ (b_ℓ_4 ∨ b_ℓ_5). The set S = {ℓ_2, ℓ_4, ℓ_5} induces a model of boolForm(φ), but it is not a branch of φ because M[S] is not a minimal model. It is easy to see that φ has two branches Br_1 = {ℓ_2, ℓ_4}, and Br_2 = {ℓ_2, ℓ_5}. Therefore, we see that Branches(φ) = {Br_1, Br_2}.

As our goal is to reason about the side-effects of variable assignments, the remainder of this section looks at how we can achieve this with the help of branches.

Assignment strategies. We want to statically mark some expressions as assignments, that is, pick a set A ⊆ Labs(φ). Below, we formulate the critical properties we require from such a set, which we will later call an assignment strategy.

Most obviously, we want to consider only assignment candidates:

Definition 2.4.1 A set H ⊆ Labs(φ) is homogeneous if all the labels in H are assignment candidates. Formally, H ⊆ cand(φ).

If we choose an arbitrary homogeneous set H, it might lack assignments on some branches or have multiple assignments to the same variable on others. Formally, we say that H has a covering index d ∈ N_0 if there is a branch Br ∈ Branches(φ) and a variable v' ∈ Vars'(φ) for which d = |Br ∩ H ∩ cand(v', φ)|. Now we define sets, that cover all branches with assignments:

Definition 2.4.2 A homogeneous set C is a covering of φ, if it does not have 0 as a covering index. It is a minimal covering of φ, if it only has 1 as a covering index.

Consider the TLA+ formula x' = y' ∧ y' = 2x'. Its corresponding α-TLA+ expression ℓ_0 :: (ℓ_1 :: x' ∈ ℓ_2 :: *(y') ∧ ℓ_3 :: y' ∈ ℓ_4 :: *(x')) has a minimal covering {ℓ_1, ℓ_3}.
However, there is no way to order the assignments to \( x' \) and \( y' \). To detect such cases, we define acyclic sets:

**Definition 2.4.3** A homogeneous set \( A \) is acyclic, if there is a strict total order \( \prec_A \) on \( A \), with the following property: For every variable \( v' \in V \), every branch \( Br \in \text{Branches} \) and every pair of labels \( \ell_i \) and \( \ell_j \) in \( A \cap Br \) the relation \( \ell_i \prec_v \ell_j \) implies \( \ell_i \prec_A \ell_j \).

Having defined homogeneous, minimal covering, and acyclic sets, we can formulate the notion of an assignment strategy.

**Definition 2.4.4** Let \( \phi \) be an \( \alpha\text{-TLA}^+ \) expression. A set \( A \subseteq L \) is an assignment strategy for \( \phi \), if it is an acyclic minimal covering.

**Static assignment problem.** Given an \( \alpha\text{-TLA}^+ \) expression \( \phi \), our goal is to find an assignment strategy, or prove that none exists.

### 2.5 Finding Assignment Strategies with SMT

For a given \( \alpha\text{-TLA}^+ \) expression \( \phi \), we construct an SMT formula \( \theta(\phi) \), that encodes the properties of assignment strategies. Technically, \( \theta(\phi) \) is defined as \( \theta_H(\phi) \land \theta_C(\phi) \land \theta_A(\phi) \), and consists of:

1. A Boolean formula \( \theta_H(\phi) \), that encodes homogeneity.
2. A Boolean formula \( \theta_C(\phi) \), that encodes the minimal covering property.
3. A formula \( \theta_A(\phi) \), that encodes acyclicity. This formula requires the theories of linear integer arithmetic and uninterpreted functions (\( QF_{UFLIA} \)).

In the following, Propositions 2.5.1–2.5.4 formally establish the relation between \( \phi \) and its three SMT counterparts. Together, the propositions allow us to prove the following theorem:

**Theorem 2.5.1** For every \( \alpha\text{-TLA}^+ \) formula \( \phi \) and \( A \subseteq \text{Labs}(\phi) \), it holds that \( M[A] \models \theta(\phi) \) if and only if \( A \) is an assignment strategy for \( \phi \).

Detailed proofs of Theorem 2.5.1 and Propositions 2.5.1–2.5.4 can be found in [KTK20].

#### 2.5.1 Homogeneous Sets

We introduce a Boolean formula, whose models are exactly those induced by homogeneous sets. To this end, take the set of labels corresponding to expressions that are not assignment candidates, \( \mathcal{N}(\phi) \), given by \( \mathcal{N}(\phi) := \text{Labs}(\phi) \setminus \text{cand}(\phi) \). Then, we define the following:

\[
\theta_H(\phi) := \bigwedge_{\ell \in \mathcal{N}(\phi)} \neg b_\ell
\]
Proposition 2.5.1 For every $\alpha$-TLA expression $\phi$ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] = \theta_H(\phi)$ if and only if $A$ is homogeneous.

2.5.2 Minimal Covering Sets

Next we construct a Boolean formula $\theta^*_C(\phi)$, whose models are exactly those induced by covering sets. To this end, we define, for each $v' \in \text{Vars}'(\phi)$, the transformation $\delta_{v'}$ as shown in Table 2.4. Intuitively, $\delta_{v'}(\phi)$ is satisfiable exactly when there is an assignment to $v'$ on every branch of $\phi$. We then define

$\theta^*_C(\phi) := \bigwedge_{v' \in \text{Vars}'(\phi)} \delta_{v'}(\phi)$

Formally, the following proposition holds:

Proposition 2.5.2 For every $\alpha$-TLA expression $\phi$ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] = \theta_H(\phi) \land \theta^*_C(\phi)$ if and only if $A$ is a covering set for $\phi$.

It is easy to restrict coverings to the minimal coverings. To do this, we define the set of collocated labels, denoted Colloc($\phi$), as

$$\text{Colloc}(\phi) := \{(\ell_1, \ell_2) \in \mathcal{L}^2 \mid \exists Br \in \text{Branches}(\phi) \cdot \{\ell_1, \ell_2\} \subseteq Br\}$$

We can use this set to reason about minimal coverings: A minimal covering may contain, per variable, no more than one label from each pair of collocated assignments to that variable. We describe these labels by using the sets $\text{Colloc}_{v'}(\phi) := \text{Colloc}(\phi) \cap \text{cand}(v', \phi)^2$ and

$$\text{Colloc}_{\text{Vars}'}(\phi) := \bigcup_{v' \in \text{Vars}'(\phi)} \text{Colloc}_{v'}(\phi)$$

Then, the following SMT formula, in addition to $\theta^*_C(\phi)$, helps us find minimal covering sets:

$$\theta^M(\phi) := \bigwedge_{(i,j) \in \text{Colloc}_{\text{Vars}'}(\phi)} \neg(b_i \land b_j)$$
We denote by $\theta_C(\phi)$ the formula $\theta_C^*(\phi) \land \theta^3(\phi)$.

**Proposition 2.5.3** For every $\alpha$-TLA$^+$ expression $\phi$ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] \models \theta_H(\phi) \land \theta_C(\phi)$ if and only if $A$ is a minimal covering set for $\phi$.

### 2.5.3 Acyclic Assignments

The last step is reasoning about acyclicity. Recall that, for $\ell_1, \ell_2 \in \mathcal{L}$, the relation $\ell_1 \prec_{\omega} \ell_2$ holds if and only if $\ell_1 \in \text{cand}(v', \phi) \land v' \in \text{frozen}_\omega(\ell_2)$. It is prudent to see that $\prec_{\omega}$ is not, in general, a strict total order (possibly not even irreflexive). However, the acyclicity property states that we can find a strict total order, which agrees with all relations $\prec_{\omega}$, on all branches.

Take $\text{Colloc}(\phi)$ to be the filtering of $\text{Colloc}(\phi)$ by the relations $\prec_{\omega}$, i.e. the set $\{(i, j) \in \text{Colloc}(\phi) \cap \text{cand}(\phi)^2 \mid \exists v' \in \text{Vars}(\phi) . i \prec_{\omega} j\}$. The SMT formula describing acyclicity is as follows:

$$\theta_A^*(\phi) := \bigwedge_{(i, j) \in \text{Colloc}_\omega(\phi)} b_i \land b_j \Rightarrow R(i) < R(j)$$

where $R$ is an uninterpreted $\mathcal{L} \rightarrow \mathbb{N}$ function, capturing assignment order. In practice, we take $\mathcal{L} = \mathbb{N}$. Unlike the previous formulas, $\theta_A^*(\phi)$ extends beyond Boolean logic, requiring both linear integer arithmetic and uninterpreted functions. Thus, a model for $\theta_A^*(\phi)$ is a pair $(M, r)$, where $M$ models the Boolean part of the formula, i.e. assigns truth values to each $b_i$, and $r : \mathbb{N} \rightarrow \mathbb{N}$ is the interpretation of $R$.

To simplify the analysis, we force $R$ to be injective, when it is restricted to $\text{Labs}(\phi)$. Otherwise, we could always construct an injective function from $R$, which respects the required inequalities. The formula we therefore consider is as follows:

$$\theta_A(\phi) := \theta_A^*(\phi) \land \bigwedge_{\ell_i, \ell_j \in \text{Labs}(\phi)} R(\ell_i) \neq R(\ell_j)$$

**Proposition 2.5.4** For every $\alpha$-TLA$^+$ expression $\phi$ and $A \subseteq \text{Labs}(\phi)$, there is a function $r : \mathbb{N} \rightarrow \mathbb{N}$, for which $(\mathcal{M}[A], r) \models \theta_H(\phi) \land \theta_A(\phi)$ if and only if $A$ is acyclic.

### 2.6 Soundness of our Approach

In this section, we show the relation between assignment strategies and the original TLA$^+$ formulas. To this end, we introduce the notion of a slice. Together, branches allow us to rewrite a TLA$^+$ formula into an equivalent disjunction of slices. Detailed proofs can be found in the journal paper [KTK20].

In TLA$^+$, there are two kinds of variables: rigid variables that correspond to the variables declared with $\text{CONSTANT}$, and flexible variables whose values change during the course of an execution. Primed versions of the variables exist only for flexible variables and are used in transition formulas. Transition formulas in TLA$^+$ are first-order terms and formulas
with flexible variables (unprimed and primed ones). We give the necessary definitions of TLA+ semantics, whereas details can be found in [Mer08]. An interpretation $\mathcal{I}$ defines a universe $|\mathcal{I}|$ of values and interprets each function symbol by a function and each predicate symbol by a relation. A state $s$ is a mapping from unprimed flexible variables to values, and a state $s'$ is a similar mapping for primed variables. A valuation $\xi$ is a mapping from rigid variables to values. Given an interpretation $\mathcal{I}$, a pair of states $(s, s')$, and a valuation $\xi$, the semantics of a TLA+ transition formula $E$ is the standard predicate logic semantics of $E$ with respect to the extended valuation of $s$, $s'$, $\xi$. With these definitions, $M = (\mathcal{I}, \xi, s, s')$ is a model for $E$, if $E$ is equivalent to true under $M$.

Let $\phi$ be a formula and $S \subseteq \text{Labs}(\phi)$. We define $\phi$ sliced by $S$, denoted slice$(\phi, S)$ in Table 2.5.

Table 2.5: The definition of slice$(\phi, S)$

<table>
<thead>
<tr>
<th>$\alpha$-TLA+ formula $\phi$</th>
<th>slice$(\phi, S)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\ell :: \top$</td>
<td>$\ell :: \top$</td>
</tr>
<tr>
<td>$\ell :: \ast(v_1', \ldots, v_i')$ or $\ell :: \nu' \in \phi_1$</td>
<td>${ \phi \ ; \ell \in S$</td>
</tr>
<tr>
<td>$\ell :: \bigwedge_{i=1}^{n} \phi_i$</td>
<td>$\ell :: \bigwedge_{i=1}^{n} \text{slice}(\phi_i, S)$</td>
</tr>
<tr>
<td>$\ell :: \bigvee_{i=1}^{n} \phi_i$</td>
<td>$\ell :: \bigvee_{i=1}^{n} \text{slice}(\phi_i, S)$</td>
</tr>
<tr>
<td>$\ell :: \exists x \in \phi_1 : \phi_2$</td>
<td>$\ell :: \exists x \in \phi_1 : \text{slice}(\phi_2, S)$</td>
</tr>
<tr>
<td>$\ell :: \text{ITE}(\phi_1, \phi_2, \phi_3)$</td>
<td>$\ell :: \text{ITE}(\phi_1, \text{slice}(\phi_2, S), \text{slice}(\phi_3, S))$</td>
</tr>
</tbody>
</table>

Below, we show that the branches and slices induced by them naturally decompose a TLA+ formula. Let $\phi$ be an $\alpha$-TLA+ expression and $\gamma(\phi)$ its corresponding TLA+ formula. Then, the following holds:

**Proposition 2.6.1** Let $\phi$ be an $\alpha$-TLA+ expression and $M = (\mathcal{I}, \xi, s, s')$ a model of the TLA+ formula $\gamma(\phi)$. There exists a branch $\text{Br}$ of $\phi$ such that $M$ is also a model of $\gamma(\text{slice}(\phi, \text{Br}))$.

**Proposition 2.6.2** Let $\phi$ be an $\alpha$-TLA+ expression and $M = (\mathcal{I}, \xi, s, s')$ a model of the TLA+ formula $\gamma(\text{slice}(\phi, \text{Br}))$. Then, $M$ is also a model of $\gamma(\phi)$.

**Proposition 2.6.3** Let $\phi$ be an $\alpha$-TLA+ expression. For every $S, T \subseteq \text{Labs}(\phi)$, every model $M$ of the TLA+ formula $\gamma(\text{slice}(\phi, S))$, is also a model of $\gamma(\text{slice}(\phi, S \cup T))$.

It is easy to see that Proposition 2.6.3 does not hold in the other direction. For instance, take the empty set as $S$ and $\text{Labs}(\phi)$ as $T$. This implies the following:

$$\gamma(\text{slice}(\phi, S)) = \top$$

Obviously, $\top$ cannot have a model, regardless of whether $\gamma(\phi)$ has one or not.
Since Propositions 2.6.1 and 2.6.2 hold, it would suffice to consider the set Branches(φ), together with an assignment strategy, to generate symbolic transitions. However, it is often the case that, for two distinct branches Br1 and Br2, the same assignments in A are chosen, that is, the intersections Br1 ∩ A and Br2 ∩ A are the same. We show that one can reduce the number of considered symbolic transitions, by analyzing how various branches intersect A.

An assignment strategy A naturally defines an equivalence relation ∼A on Branches(φ), given by Br1 ∼A Br2 if and only if Br1 ∩ A = Br2 ∩ A. We use the notation [Br]A to refer to the equivalence class of Br by ∼A, that is, the set \{X ∈ Branches(φ) | Br ∼A X\}.

**Definition 2.6.1** Let φ be an α-TLA+ expression, A an assignment strategy for φ and Br a branch of φ. Using X = [Br]A and Y = \bigcup_{Z ∈ X} Z, we define the symbolic transition generated by Br and A to be slice(φ, Y).

**Remark 2.6.1.** Let us look Example 2.4.1 again. The formula φ has two assignment strategies A1 = {ℓ2}, and A2 = {ℓ4, ℓ5}. If the first assignment strategy A1 is chosen, we have that Br1 ∩ A1 = Br2 ∩ A1 = {ℓ2}. This implies that Br1 and Br2 are in the same equivalence class, or Br1 ∼A1 Br2. Therefore, we have only one symbolic transition which is exactly φ. However, if A2 is selected, branches Br1 and Br2 are not equivalent because Br1 ∩ A2 = {ℓ4} and Br2 ∩ A2 = {ℓ5}. Therefore, we have two symbolic transitions:

\[ T_1 = ℓ_1 :: \left[\ell_2 :: x' ∈ *\right] \land \left[\ell_3 :: \left[\ell_4 :: x' ∈ * \lor \ell_5 :: F\right]\right] \]
\[ T_2 = ℓ_1 :: \left[\ell_2 :: x' ∈ *\right] \land \left[\ell_3 :: \left[\ell_4 :: F \lor \ell_5 :: x' ∈ *\right]\right] \]

The first assignment strategy A1 seems to be better than A2 because A1 generates fewer symbolic transitions than A2. However, in this chapter, we do not introduce any metric, by which we could compare assignment strategies. In the implementation, we use any strategy found by the SMT solver.

The equivalence relation ∼A allows us to define a counterpart to Proposition 2.6.3.

**Proposition 2.6.4** Let φ be an α-TLA+ expression. For any selection Br1, . . . , Brk from the branches of φ, the following holds: If there exists a model M of the formula γ(slice(φ, Br1 ∪ · · · ∪ Brk)), then M must be a model of γ(slice(φ, Br)), for some branch Br ∈ Branches(φ). Additionally, if there is an assignment strategy A for φ, such that Br1, . . . , Brk all belong to the same equivalence class [B]A, then M must be a model of γ(slice(φ, Br)), for some branch Br ∈ [B]A.

The following result allows us to use symbolic transitions, not individual branches:

**Theorem 2.6.1** Let φ be an α-TLA+ expression and A an assignment strategy for φ. There is a model M of the TLA+ formula γ(φ) if and only if there exists a Br ∈ Branches(φ), such that M is a model of γ(ψ), where ψ is the symbolic transition generated by Br and A.
2.7 Experiments and Potential Applications

Implementation and evaluation. Based on the theory presented in this chapter, we have implemented a procedure to find assignment strategies and their corresponding symbolic transitions from TLA+ specifications, or report that none exist. It uses Z3 as the background SMT solver.

We have chosen specifications both from a collection of algorithms we have encoded in TLA+ ourselves and from publicly available sources, e.g. EWD840 and Paxos from the repository TLA+ Examples. For each specification, we focus on the Next formula. We report the number of subexpressions in α(Next), that is, |Sub(α(Next))|, the number of assignments in the strategy found by our procedure, the number of symbolic transitions computed and the total runtime. The results are presented in Table 2.6. Note that the results for the specification in Fig. 2.1 are as expected; all assignment candidates must be part of the strategy and we find two symbolic transitions corresponding to Produce and Consume. We also see that the number of symbolic transitions is generally much smaller than the number of transitions an explicit-state model checker would find, as even simple specifications, like in Figure 2.1, would generate numerous transitions in explicit state model checking, but only two symbolic transitions.

Applications. We illustrate an application of our technique for bounded model checking [BCCZ99] by the means of the example in Figure 2.3. In this example, three processes pass a unique token in one direction, with the goal of computing the largest process identifier.

Our technique extracts three symbolic transitions $T_1$, $T_2$, and $T_3$, each $T_i$ being equivalent to $P(i) \land id' = id$ for $1 \leq i \leq 3$. As common in bounded model checking, with $[[F]]_{i,i+1}$ we denote the SMT encoding of a transition by action $F$ from an $i$th to an $(i+1)$-th state. For instance, $[[Next]]_{0,1}$ and $[[T_3]]_{0,1}$ encode the transitions from the state 0 to the state 1 by Next and $T_3$. Likewise, $[[Init]]_0$ encodes SMT constraints by Init on the initial states. One can use the SMT encodings introduced in [MV12a, MV12b].

Figure 2.4 shows the SMT formulas that are constructed by a bounded model checker when exploring executions up to length 4. (For the sake of space, we omit the formulas.

---

Figure 2.3: A distributed maximum computation in a ring of three processes in TLA+

---

EXTENDS Naturals

VARIABLE tok, max, id

Init $\triangleq$ tok $= 1 \land id \in [1..3 \rightarrow \text{Nat}] \land max = 0$

$P(i) \triangleq$ tok $= i \land tok' = 1 + i \% 3 \land max' = \text{IF } id[i] > max \text{ THEN } id[i] \text{ ELSE } max$

Next $\triangleq (P(1) \lor P(2) \lor P(3)) \land id' = id$
2. Extracting Symbolic Transitions from TLA$^+$ specifications

Table 2.6: Experimental results. The meaning of columns is as follows: “#expr” is the number of expressions, “#assign” is the size of the assignment strategy (number of assignments), “#trans” is the number of constructed symbolic transitions.

<table>
<thead>
<tr>
<th>Specification</th>
<th>#expr</th>
<th>#assign</th>
<th>#trans</th>
<th>time [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aba_asyn_byz</td>
<td>184</td>
<td>48</td>
<td>8</td>
<td>91</td>
</tr>
<tr>
<td>AlternatingBit</td>
<td>183</td>
<td>49</td>
<td>7</td>
<td>169</td>
</tr>
<tr>
<td>Bakery</td>
<td>303</td>
<td>71</td>
<td>16</td>
<td>260</td>
</tr>
<tr>
<td>BcastByz</td>
<td>96</td>
<td>22</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td>BcastFolklore</td>
<td>75</td>
<td>17</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>Bosco</td>
<td>114</td>
<td>18</td>
<td>9</td>
<td>29</td>
</tr>
<tr>
<td>Boulanger</td>
<td>353</td>
<td>85</td>
<td>18</td>
<td>277</td>
</tr>
<tr>
<td>C1cs</td>
<td>171</td>
<td>37</td>
<td>8</td>
<td>76</td>
</tr>
<tr>
<td>Cbc_max</td>
<td>265</td>
<td>72</td>
<td>9</td>
<td>142</td>
</tr>
<tr>
<td>CF1s_folklore</td>
<td>258</td>
<td>69</td>
<td>14</td>
<td>139</td>
</tr>
<tr>
<td>ChangRoberts</td>
<td>88</td>
<td>18</td>
<td>7</td>
<td>56</td>
</tr>
<tr>
<td>Channel</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>142</td>
</tr>
<tr>
<td>DieHard</td>
<td>43</td>
<td>12</td>
<td>6</td>
<td>28</td>
</tr>
<tr>
<td>DijkstraMutex</td>
<td>305</td>
<td>75</td>
<td>18</td>
<td>271</td>
</tr>
<tr>
<td>EWDS84</td>
<td>79</td>
<td>16</td>
<td>4</td>
<td>47</td>
</tr>
<tr>
<td>HourClock</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>19</td>
</tr>
<tr>
<td>LamportMutex</td>
<td>130</td>
<td>30</td>
<td>6</td>
<td>75</td>
</tr>
<tr>
<td>MissionariesAndCannibals</td>
<td>20</td>
<td>2</td>
<td>1</td>
<td>44</td>
</tr>
<tr>
<td>Nbacc_ray97</td>
<td>77</td>
<td>15</td>
<td>14</td>
<td>48</td>
</tr>
<tr>
<td>Nbacg_guer01</td>
<td>296</td>
<td>82</td>
<td>13</td>
<td>220</td>
</tr>
<tr>
<td>Paxos</td>
<td>92</td>
<td>16</td>
<td>4</td>
<td>49</td>
</tr>
<tr>
<td>PaxosCommit</td>
<td>153</td>
<td>28</td>
<td>10</td>
<td>179</td>
</tr>
<tr>
<td>Prisoners</td>
<td>60</td>
<td>14</td>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>Queens</td>
<td>19</td>
<td>4</td>
<td>2</td>
<td>34</td>
</tr>
<tr>
<td>Raft</td>
<td>841</td>
<td>222</td>
<td>23</td>
<td>932</td>
</tr>
<tr>
<td>SchedulingAllocator</td>
<td>73</td>
<td>12</td>
<td>4</td>
<td>19</td>
</tr>
<tr>
<td>SimpleAllocator</td>
<td>40</td>
<td>6</td>
<td>3</td>
<td>35</td>
</tr>
<tr>
<td>Spanning</td>
<td>74</td>
<td>12</td>
<td>4</td>
<td>27</td>
</tr>
<tr>
<td>TCommit</td>
<td>24</td>
<td>3</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>TwoPhase</td>
<td>122</td>
<td>28</td>
<td>7</td>
<td>65</td>
</tr>
<tr>
<td>Voting</td>
<td>39</td>
<td>4</td>
<td>2</td>
<td>25</td>
</tr>
</tbody>
</table>
2.8 Related Work

![Diagram showing SMT formulas constructed for executions up to length 4: using the action Next (left) and using symbolic transitions (right). The gray formulas are excluded from the SMT context during the exploration.]

Table 2.7: TLA+ expressions that may be rejected by either TLC or the presented technique

<table>
<thead>
<tr>
<th>No.</th>
<th>Variables</th>
<th>Expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x, y$</td>
<td>$\text{ITE}(x = 1, y' = 1, x' = 1 \land y' = 1)$</td>
</tr>
<tr>
<td>2</td>
<td>$x, y$</td>
<td>$x' = y' \land y' = x' \land y' = 1$</td>
</tr>
</tbody>
</table>

that check property violation.) On one hand, the monolithic encoding that uses only Next has to keep all the formulas in the SMT context. On the other hand, by incrementally checking satisfiability of the SMT context, the model checker can discover that some formulas — for example, $[T_2]_{0,1}$ and $[T_3]_{1,2}$ — lead to unsatisfiability and prune them from the SMT context. A similar approach improves efficiency of bounded model checking C programs [BHvM09] [Ch. 16], hence, we expect it to be effective for the verification of TLA+ specifications too.

2.8 Related Work

Since our approach and TLC (version 1.6.0) follow different strategies\(^2\), the user might receive incompatible outcomes for unusual transition predicates. In the following, we discuss examples given in Table 2.7:

1. Assume that $x = 0$ in the first example. In this case, TLC never goes in the THEN arm, and therefore, it does not produce an error. Our tool reports that there is no assignment strategy for the missing $x'$ in the THEN arm.

2. TLC complains that “the identifier $y$ is either undefined or not an operator”, because TLC evaluates expressions from left to right. In contrast, our approach finds an assignment strategy, and considers $y' = 1$ and $x' = y'$ to be variable assignments.

A new tool for extracting symbolic transitions in TLA+ has been introduced for APALACHE in [Inf]. This tool considers the left-to-right syntax order of and-operator ($\land$)

\(^2\)TLC evaluates formulas in a fixed order: from top to bottom and from left to right.
arguments and requires that a primed variable must be assigned before it is used. Consider an expression $A \triangleq x' > 0 \land x' = 1$. While our approach considers $A$ as a symbolic transition, the tool in \cite{Inf} throws an exception as the expression $x' > 0$ precedes any assignment to $x$.

For the translation from TLA$^+$ to B in \cite{HL12}, the authors introduced an algorithm to extract actions in TLA$^+$. While an expression $x' = 0 \land (y' = 1 \lor y' = 0)$ may be considered as one action in \cite{HL12}, our approach slices this expression into two symbolic transitions. Moreover, their approach is not based on an SMT encoding.

### 2.9 Summary

We have introduced a technique for computing symbolic transitions of a TLA$^+$ specification by finding expressions that can be interpreted as assignments. Importantly, we designed the technique with soundness in mind. Detailed proofs can be found in the journal paper \cite{KTK20}. Our results are implemented as a first preprocessing step in the symbolic model checker APALACHE for TLA$^+$.

As in the case of TLC, one can find TLA$^+$ specifications, for which no assignment strategy exists. However, TLA$^+$ users are systematically checking their specifications with TLC, in order to find errors. Hence, it is rarely the case in practice that a specification is checked with TLC but not checked with APALACHE.
TLA+ Model Checking Made Symbolic

TLA+ is extremely concise yet expressive: The language primitives include Booleans, integers, functions, tuples, records, sequences, and sets thereof, which can be also nested. This is probably why the only model checker for TLA+ (called TLC) relies on explicit enumeration of values and states.

In this chapter, we present APALACHE - a first symbolic model checker for TLA+. Unlike TLC, it assumes that all specification parameters are fixed and all states are finite structures. Unlike TLC, APALACHE translates the underlying transition relation into quantifier-free SMT constraints, which allows us to exploit the power of SMT solvers. Designing this translation is the central challenge that we address in this chapter. Our experiments show that APALACHE outperforms TLC on examples with large state spaces.

This chapter adapts the paper presented at OOPSLA’19 [KKT19]. Since this paper was published, the latest version of APALACHE 0.16.2 was used by verification engineers at Informal Systems to check blockchain protocols, e.g., Tendermint blockchain synchronization and a Tendermint light client.

3.1 Our Approach at a Glance

In Section 1.1 we discuss challenges in developing a general-purposed symbolic model checker for TLA+. Satisfiability-modulo-theory (SMT) solvers such as Z3 [DB08] are efficient tools to reason about logical constraints. In the following we present a brief introduction to our approach that utilizes the power of state-of-the-art SMT solvers.

We are developing a symbolic model checker that is powered by a satisfiability-modulo-theory (SMT) solver such as Microsoft Z3 [DB08]. To make the tool usable for the TLA+
community, we aim at introducing as few restrictions to the language as TLC does. Hence, whenever we have a choice between an efficient SMT encoding that restricts the input and a less efficient but general SMT encoding, we choose the general one. (Indeed, we plan optimizations for the special fragments of TLA+ in the future.) Similar to TLC, we make several pragmatic assumptions about the input specifications:

1. All input parameters are fixed. Although TLA+ specifications are typically parameterized, the users restrict parameters to run TLC.

2. Reachable states and the values of the parameters are finite structures, e.g., finite sets and functions of finite domains. This is also a requirement of TLC. (The latest version of APALACHE supports integer constants.)

3. Following our work in Chapter 2, we assume that for each variable \( x \), there is a set of expressions \( x' = e \) and \( x' \in S \) that can be treated as assignments to \( x' \). As a consequence, the specification can be decomposed into a set of symbolic transitions.

4. The specification is well-typeable in our type system.

The main challenge of this chapter comes from the expressiveness of TLA+. Among basic types, it supports Booleans, integers, and uninterpreted constants. Among structured types, it supports sets, functions, tuples, records, and sequences; all of them can be arbitrarily nested in each other. Moreover, it is common to use power sets, sets of functions, and set cardinalities in TLA+ specifications. Multiple techniques were developed for sets and cardinalities in SMT [KNR05, YPK10, DHV+14, vGBR16, TRBB18, BLL+19, CR16]. Although these techniques can be used to reason about some TLA+ expressions, they pose various constraints on the set theory that would not easily accommodate typical TLA+ specifications. [MV18] introduced an unsorted SMT encoding of TLA+ for discharging proof obligations in TLAPS. This encoding did not scale to model checking in our preliminary experiments. Hence, we introduce a multi-sorted encoding.

**Contributions.** Our main contributions in this chapter are as follows:

1. We introduce the kernel fragment \( \text{KerA}^+ \) to capture all but few TLA+ operators over finite structures.

2. We define operational semantics of \( \text{KerA}^+ \) in terms of reduction rules. Given a \( \text{KerA}^+ \) formula \( \phi \), the reduction system produces SMT constraints that are equisatisfiable to \( \phi \).

3. We show how to use the reduction system for: (a) checking inductive invariants, and (b) checking safety of TLA+ specifications by bounded model checking.
3.2. Preprocessing: Flattening, Assignments, and Types

3.2.1 Flattening

As exemplified by Section 1.1, TLA+ specifications are normally written as a collection of operator definitions. They can be also organized in modules. As the operator \textit{Next} describes one step of a system execution, the operators in TLA+ are usually non-recursive. They are similar to macros in programming languages. As a first step, our technique replaces calls to the user-defined operators with the operator bodies; as expected, the formal arguments are substituted with the arguments at the call sites. The same applies to the local operators that are defined with the \texttt{LET-IN} expression. We also instantiate modules, in order to obtain a single-module specification, in which the operators \textit{Init} and \textit{Next} contain only the calls to the built-in TLA+ operators. The flattening phase is purely syntactic, so we obviously obtain an equivalent TLA+ specification.

Note on Recursive Operators. [Lam18] recently added recursive operators to TLA+ version 2. Hence, the users can conveniently write expressions in terms of recursion.

Figure 3.1 shows the main phases of APALACHE [Sys19]. First, the call sites of user-defined operators are replaced with their bodies, which produces a flat specification. Second, the technique in Chapter 2 finds symbolic transitions in the specification. Third, basic type inference labels expressions with types. Finally, the reduction system produces SMT constraints. A query to the SMT solver gives us an answer to the model checking question.

Structure. We discuss the preprocessing steps in Section 3.2. In Section 3.3, we introduce the kernel language KerA++. In Section 3.4, we provide how to define many TLA+ operators in KerA++. We introduce the reduction framework in Section 3.5 and the reduction rules in Sections 3.6-3.11. In Section 3.12, we discuss the implementation. We finish with the discussion of the experiments in Section 3.13.
3. TLA⁺ Model Checking Made Symbolic

Instead of logical formulas. As is common in bounded model checking, we could unroll a call to a recursive operator up to a bound predefined by the user, which would produce a large TLA⁺ formula. To implement an incremental unrolling, we would need an advanced type checker, which we postpone for the future.

3.2.2 Assignments and Symbolic Transitions

As noted earlier, there is no notion of variable assignment in TLA⁺. However, the model checker TLC interprets expressions \( x' = e \) and \( x' \in S \) as assignments, if \( x' \) has not been assigned a value before. TLC evaluates formulas in a fixed order: from top to bottom and from left to right. Moreover, it treats some disjunctions as non-deterministic choice.

In Chapter 2, we introduced a symbolic technique for finding such assignments without evaluating the TLA⁺ formula. Additionally, we proposed a technique for decomposing a TLA⁺ formula into a disjunction of formulas \( T_1, \ldots, T_k \) in the following way:

1. Assignment completeness: For every variable \( v \), each \( T_i \) has at least one assignment to \( v \), and
2. Single assignment: For every variable, each \( T_i \) contains exactly one assignment to it.

We apply this technique to find assignments and symbolic transitions.

Remark 3.2.1. Consider the example in Figure 1.1. There are 7 symbolic transitions, corresponding to the possible actions \( TMCommit, TMAbort, TMRcvPrepared(rm) \), and so on. The body of \( TMAbort \) contains assignments to all five variables; two of them are unchanged.

3.2.3 Types

Whereas TLA⁺ is untyped by design, TLC dynamically computes types and rejects some combinations of legal TLA⁺ expressions, e.g., \( \{1, “a”\} \). However, TLC’s type system is not defined. We use the following type system, which is similar to the type system by [MV12a]:

\[
\tau ::= \text{Name} \mid \text{Bool} \mid \text{Int} \mid \tau \rightarrow \tau \mid \text{Set}[\tau] \mid \text{Seq}[\tau] \mid \tau^* \mid \{n \tau_1, \ldots, n \tau_k \}
\]

The type system rejects some TLA⁺ expressions that are legal in the untyped language. Importantly, elements of sets must have the same type. For example, \( \{1, \{2, 3\}\} \) is ill-typed. Similarly, TLA⁺ functions can be defined on values of different types and return values of different types, but such functions are rejected by the type system. Finally, our type system clearly distinguishes between functions, sequences, tuples, and records. (This challenge has been solved in separate work [KK20].)
Developing a fully automatic type inference engine for TLA$^+$ is a challenge on its own.
In this chapter, we follow a simple approach: In most cases, the types are computed automatically by propagation; when the tool fails to find a type, it asks the user to write a type annotation. Given the syntax tree of a TLA$^+$ expression, our basic type inference algorithm works as follows:

1. A leaf expression is assigned the respective type. For instance, the literals $0, 1, -1, \ldots$ have type $\text{Int}$, and the literals $\text{false}$ and $\text{true}$ have type $\text{Bool}$. If the type is ambiguous, as in $\{\}$, then type inference fails, and the user has to annotate the expression with a type.

2. A non-leaf expression is an application of a built-in operator. The type signatures of these operators are predefined, e.g., $+: \text{Int} \times \text{Int} \to \text{Int}$. Some operators introduce bound variables, e.g., $\exists x \in S : e$ or $\{e : x \in S\}$. As expected, the type of the binding set is computed first, and then the type of $e$ is computed.

In practice, the user has only to give the types of empty sets, empty sequences, and records. It is common to mix records of different types. In Section 1.1, records $[\text{type} \mapsto \text{"Abort"}]$ and $[\text{type} \mapsto \text{"Prepared"}, \text{rm} \mapsto \text{rm}]$ are both added to the set $\text{msgs}$. The user has to annotate the records and their sets with a super type, e.g., $[\text{type} : \text{Name}, \text{rm} : \text{Set[Name]}]$.

### 3.3 KerA$^+$: the Kernel Language of TLA$^+$ Expressions

Our main goal is to check TLA$^+$ specifications using an SMT solver as a back-end. A direct translation of the rich TLA$^+$ syntax would be tedious and error-prone. Hence, we introduce KerA$^+$: A small set of operators that can express all but a few TLA$^+$ expressions. For example, it includes the operator $\text{union} \{S_1, \ldots, S_n\}$, which constructs the union $S_1 \cup \cdots \cup S_n$. The binary operator $S_1 \cup S_2$ is equivalent to $\text{union} \{S_1, S_2\}$. We add a few auxiliary operators that simplify the translation.

A list of KerA$^+$ expressions is given in Table 3.1. It might seem surprising that very basic operators such as Boolean operators are missing. In fact, they can be expressed with if-then-else:

$$\neg p \equiv \text{ITE}(p, \text{false}, \text{true}) \quad p \land q \equiv \text{ITE}(p, q, \text{false}) \quad p \lor q \equiv \text{ITE}(p, \text{true}, q)$$

Several KerA$^+$ operators do not originate from TLA$^+$:

- **Assignment $x' \in S$:** Following TLC, under the conditions given in Chapter 2, we treat an expression $x' \in S$ as an assignment of a value from the set $S$ to the variable $x'$. Note that an expression $x' = e$ is a special case of this rule, which can be written as $x' \in \{e\}$. We label such assignments with $x' \in S$, to distinguish them from membership tests $x' \in S$. 

41
Table 3.1: The language KerA+. We highlight the expressions that do not have counterparts in pure TLA+.

| Literals: | FALSE, TRUE | 0, 1, 2, ..., c1, ..., cn (constants) |
| Integers: | n1 • n2 where • is one of: +, -, *, /, %, <, ≤, >, ≥, =, ≠ |
| Sets: | {e1, ..., en} | \{x ∈ S : p\} | \{e : x ∈ S\} | UNION S |
| Control: | ITE(p, e1, e2) | e1 ⊕ ... ⊕ en | x' ∈ S | x' ∈ [S1 → S2] | x ∈ SUBSET S |
| Quantifiers: | Ǝx ∈ S : p | CHOOSE x ∈ S : p | FROM e1, ..., en BY θ |
| Functions: | [x ∈ S → e] | f[e] | DOMAIN f | [f EXCEPT ![e1] = e2] |
| Records: | [nm1 → e1, ..., nmn → en] | s[i] | DOMAIN s | [s EXCEPT ![i] = e] |
| Sequences: | (e1, ..., en) | s[0] | DOMAIN s | s[0] |

- **Non-deterministic disjunction** φ1 ⊕ ... ⊕ φn: This operator formalizes the special form of TLC disjunction. It evaluates to true if and only if the disjunction φ1 ∨ ... ∨ φn evaluates to true. However, non-deterministic disjunction adds constraints on the variable assignments: For every i, j ∈ 1..n and i ≠ j, formula φi contains an assignment to a variable x' if and only if formula φj contains an assignment to x'. Note that this property is implied by the single-assignment property of symbolic transitions (see Section 3.2.2). Hence, we use it to compose the symbolic transitions.

- **Choice with an oracle** FROM e1, ..., en BY θ: This operator returns expression ei when θ = i and 1 ≤ i ≤ n; otherwise, it returns an arbitrary value of the same type as e1, ..., en.

KerA+ is a subset of TLA+—except for the three operators discussed above—and the meaning of the operators coincides with the description in the book by [Lam02]. Denotational semantics of TLA+ in first-order logic is given by [Mer08]. In Sections 3.6–3.10 we give a brief description of each KerA+ operator along with the semantics for finite structures in terms of rewriting rules.

### 3.4 Defining TLA+ Operators in KerA+

In this section, we provide how to define many TLA+ operators in KerA+. The following definitions are detailed. Note that many TLA+ operators are not included in KerA+, e.g., ¬, ∪, ∩. This is because they can be derived from the included operators as follows.

#### 3.4.1 Logic

¬p := ITE(p, FALSE, TRUE)  
p ∧ q := ITE(p, q, FALSE)  
p ∨ q := ITE(p, TRUE, q)
3.4. Defining TLA\textsuperscript{+} Operators in \textsf{KerA}\textsuperscript{+}

\[ p \Rightarrow q := \neg p \lor q \quad p \equiv q := p \Rightarrow q \land q \Rightarrow p \quad \forall x \in S : p := \neg \exists x \in S : \neg p \]

3.4.2 Sets

\[ x \in S := \exists y \in S : y = x \quad S \cup T := \text{UNION} \{ S, T \} \quad S \cap T := \{ x \in S : x \in T \} \]
\[ S \setminus T := \{ x \in S : x \notin T \} \quad S \subseteq T := \forall x \in S : x \in T \quad S =_{\text{set}} T := S \subseteq T \land T \subseteq S \]

3.4.3 Functions

\[ [f \text{ EXCEPT } ![e_1] ![e_2] = e] := [f \text{ EXCEPT } ![e_1] = [f[e_1] \text{ EXCEPT } ![e_2] = e]] \]
\[ [f \text{ EXCEPT } ![x] = e_1, ![y] = e_2]_\rho := [[f \text{ EXCEPT } ![x] = e_1 \text{ EXCEPT } ![y] = e_2] \]

The excepts with more arguments are defined similarly.

3.4.4 Records and Tuples

\[ [nm_1 : S_1, \ldots, nm_n : S_n] := \{ [nm_1 \mapsto x_1, \ldots, nm_n \mapsto x_n] : x_1 \in S_1, \ldots, x_n \in S_n \} \]
\[ r_1 =_{\text{rec}} r_2 := \text{DOMAIN } r_1 = \text{DOMAIN } r_2 \land \forall x \in \text{DOMAIN } r_1 . r_1.x = r_2.x \]
\[ S_1 \times \cdots \times S_n := \{ (x_1, \ldots, x_n) : x_1 \in S_1, \ldots, x_n \in S_n \} \]

3.4.5 Control

\[ \text{CASE } p_1 \rightarrow e_1 \quad \cdots \quad \square p_n \rightarrow e_n \quad \square \text{OTHER } \rightarrow e := \text{ITE}(p_1, e_1, \ldots, \text{ITE}(p_n, e_n, e)) \]
\[ \text{CASE } p_1 \rightarrow e_1 \quad \cdots \quad \square p_n \rightarrow e_n := \text{ITE}(p_1, e_1, \ldots, \text{ITE}(p_n, e_n, \text{TRUE})) \]

3.4.6 Sequences

Following [Lam02], a sequence is a function from the set \([1..n \rightarrow S]\) for an integer \(n \geq 1\) and a set \(S\). Instead of defining sequences as in Section 3.9, we could exclude the sequence operators from \textsf{KerA}\textsuperscript{+} and rewrite them as follows (though it would produce a less efficient encoding):
3. TLA\(^+\) Model Checking Made Symbolic

\[(e_1, \ldots, e_n)_s \triangleq [i \in 1..n \mapsto \text{ite}(i = 1, e_1, (\ldots (\text{ite}(i = n - 1, e_{n-1}, e_n)) \ldots))]\]
\[s[i] \triangleq s[i]
\]
\[\text{DOMAIN } s \triangleq \text{DOMAIN } s\]
\[[s \text{ EXCEPT } ![i] = e] \triangleq [s \text{ EXCEPT } ![i] = e]\]
\[\text{Len}(s) \triangleq \text{FROM } \{m \in \text{DOMAIN } s: \neg (\exists i \in \text{DOMAIN } s: i > m)\}\]
\[\text{Head}(s) \triangleq s[1]
\]
\[\text{Tail}(s) \triangleq s[\text{Len}(s)]\]
\[\text{SubSeq}(s, i, j) \triangleq [\ell \in \{k \in \text{DOMAIN } s: k \leq j - i + 1\} \mapsto s[i + \ell]]\]
\[s \circ t \triangleq [i \in \text{DOMAIN } s \cup \{j \in \text{DOMAIN } t \mapsto \text{Len}(s) + j\} \mapsto \text{ite}(i \leq \text{Len}(s), s[i], t[i - \text{Len}(s)])]\]

3.5 Rewriting Framework

Our goal is to translate a KER\(^+\) expression into an equisatisfiable quantifier-free SMT formula. To this end, we introduce an abstract reduction system that allows us to iteratively transform a KER\(^+\) expression by applying reduction rules. The central idea of our approach to rewriting is to construct an overapproximation of the data structures with a graph whose edges connect values such as sets and their elements. We call this graph an arena, as it resembles the in-memory data structures that are created by the explicit-state model checker TLC. While some rules for KER\(^+\) operators extend the arena with new nodes and edges, other rules use this graph to produce SMT constraints on the actual values. The reduction rules collapse a complex KER\(^+\) expression into a so-called cell that captures the result of symbolically evaluating the expression. The rewriting process terminates, when the input KER\(^+\) formula \(\phi\) has been collapsed to a single cell. In this case, the reduction rules have produced a set of SMT constraints that are equisatisfiable to the formula \(\phi\).

3.5.1 Cells

In our framework, a cell is simply a first-order constant that is annotated with a type \(\tau\). The cells of types Int and Bool are interpreted in SMT as integers and Booleans respectively, whereas the cells of the other types remain uninterpreted. In the following, we use notation \(c_i\) or \(c_{\text{name}}\) to refer to a cell. We assume fixed a finite set of cells \(C\), which contains sufficiently many elements for rewriting a KER\(^+\) expression.

New cells are introduced when rewriting a KER\(^+\) expression. For example, the expression \(\{1, 2\}\) is rewritten by a series of rewriting steps: \(\{1, 2\} \leadsto \{c_1, 2\} \leadsto \{c_1, c_2\} \leadsto c_3\). We give the precise definition of \(\leadsto\) in Section 3.5.4. While the original expression does not contain cells, the rewritten expressions do. In fact, cells are well-formed KER\(^+\) expressions, as they can be seen as KER\(^+\) constants. Hence, the introduced cells can be seen as: (1) first-order constants in SMT, and (2) KER\(^+\) constants in KER\(^+\), which would be introduced in TLA\(^+\) using the string notation, e.g., “abc”.

44
### 3.5.2 Arenas

An arena is a directed acyclic labelled graph $\mathcal{A} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} \subseteq \mathcal{C}$ is a finite set, called arena cells, and $\mathcal{E} \subseteq \mathcal{V} \times (1..|\mathcal{V}|) \times \mathcal{V}$ is a relation between the cells, called arena edges, that have the following properties:

1. There are no duplicate labels. Formally, for every pair $(v_1, i_1, w_1), (v_2, i_2, w_2) \in \mathcal{E}$, if $v_1 = v_2$ and $w_1 \neq w_2$, then $i_1 \neq i_2$.

2. There are no gaps in the labels. Formally, for every $(v, i, w) \in \mathcal{E}$, and every index $j \in 1..(i - 1)$, there is a cell $w \in \mathcal{V}$ with the property $(v, j, w) \in \mathcal{E}$.

We write $\mathcal{V}(\mathcal{A})$ and $\mathcal{E}(\mathcal{A})$ to refer to the cells and edges of arena $\mathcal{A}$ respectively. With $c_1 \xrightarrow{i} \mathcal{A} c_2$, we denote that $(c_1, i, c_2) \in \mathcal{E}$. Similarly, we write $c \rightarrow \mathcal{A} c_1, \ldots, c_n$ to say that $c$ points to $c_1, \ldots, c_n$ in this order, that is, $c \xrightarrow{1} \mathcal{A} c_i$ for $1 \leq i \leq n$ and for every $c' \in \mathcal{V}(\mathcal{A})$ and $j > n$, it holds that $(c, j, c') \notin \mathcal{E}(\mathcal{A})$. We use the following notation to extend an arena $\mathcal{A}$:

- Notation $\mathcal{A}, c : \tau$ to introduce the arena $(\mathcal{V}', \mathcal{E}')$ such that $\mathcal{V}' = \mathcal{V}(\mathcal{A}) \cup \{c\}$ and $\mathcal{E}' = \mathcal{E}(\mathcal{A})$, provided that $c$ is a fresh cell of type $\tau$, i.e., $c \notin \mathcal{V}(\mathcal{A})$.

- Notation $\mathcal{A}, c \rightarrow c_1, \ldots, c_n$ to introduce the arena $\mathcal{A}'$ such that $\mathcal{V}(\mathcal{A}') = \mathcal{V}(\mathcal{A})$ and $\mathcal{E}(\mathcal{A}') = \mathcal{E}(\mathcal{A}) \cup \{(c, i, c_i) \mid 1 \leq i \leq n\}$.

**Remark 3.5.1.** Figure 3.2 shows examples of memory arenas for several KERA$^+$ expressions. In example (a), the arena contains six cells: three cells of type $\text{Int}$ that represent integers 1, 2, 3; two cells of type $\text{Set}[\text{Int}]$ that represent the sets $\{1, 2\}$ and $\{2, 3\}$; and one cell of type $\text{Set}[\text{Set}[\text{Int}]]$ that represents the set of sets $\{\{1, 2\}, \{2, 3\}\}$. Importantly, the arena only gives us a static overapproximation of the set. The actual contents of the set encoded by cell $c_6$ may be $\emptyset$ or $\{\{1\}, \{2\}\}$. The further constraints on the cell contents are encoded in SMT, see Section 3.5.3.

In example (b), the arena contains five cells: three cells to encode the integers, the cell $c_{14}$ to encode the record $[b \mapsto 0, c \mapsto 3]$, and the cell $c_{15}$ to encode the tuple $\langle \text{“a”}, 3, [b \mapsto 0, c \mapsto 3] \rangle$. In case of tuples, the cell type gives us unambiguous relation between the tuple fields and the cells pointed by the cell. For instance, from the edge $c_{15} \xrightarrow{1} c_{11}$ and the tuple type $\text{Name * Int * [Int, Int]}$, we immediately obtain that cell $c_{11}$ is the first field of the tuple $c_{15}$. The same applies to records.

Finally, example (c) shows the arena that is constructed for the function $f = [x \in \{1, 2\} \mapsto 1 + x]$. In our encoding, a function $f$ is represented with its relation, that is, the set $\{(x, f(x)) : x \in \text{DOMAIN} f\}$. Hence, the cells $c_{21}, c_{22},$ and $c_{23}$ encode the integers 1, 2, and 3 respectively. The cells $c_{24}$ and $c_{25}$ encode the pairs $\langle 1, 2 \rangle$ and $\langle 2, 3 \rangle$ of the relation respectively. The cell $c_{26}$ encodes the function relation, which is pointed by the function.
3. TLA⁺ Model Checking Made Symbolic

Figure 3.2: Examples of arenas for data structures in Kera⁺. The leaf cells are equal to the following constants: \( c_1 = c_{21} = 1 \), \( c_2 = c_{22} = 2 \), \( c_3 = c_{23} = 3 \), \( c_{11} = a \), \( c_{12} = 3 \), and \( c_{13} = 0 \)

cell \( c_{27} \). While the function cell \( c_{27} \) may look redundant in the presence of the cell \( c_{26} \), we keep the both, as they have different types.

Although the values of leaf cells are fixed in our examples, they do not have to be. In example (c) we could leave the values of the cells \( c_{21}, c_{22}, \) and \( c_{23} \) unconstrained. Then, the SMT solver would find values that satisfy the symbolic constraints such as \( c_{22} = 1 + c_{21} \), as prescribed by the function \( f \).

3.5.3 SMT Constraints

We recapitulate the necessary notions related to many-sorted first-order logic. We assume fixed a set of sorts \( S \), which includes exactly one sort \( s_\tau \) per type \( \tau \) that is defined in Section 3.2. Further, let \( F \) be a set of functional symbols, each functional symbol is assigned a non-negative arity. For convenience, we say that the set of cells \( C \) coincides with the set of functional symbols of arity 0 from the set \( F \). Each symbol \( f \in F \) is assigned a sort \( \text{sort}(f) \in S \). The ground terms are defined as follows: (1) every constant \( c \in C \) is a ground term, and (2) if \( t_1, \ldots, t_n \) are ground terms and \( f \in F \) has arity \( n \), then \( f(t_1, \ldots, t_n) \) is a ground term, if the sorts of \( f, t_1, \ldots, t_n \) are compatible.

We distinguish the set of predicates \( P \subseteq F \), which contains the symbols that are assigned a sort \( s_\tau_1 \times \cdots \times s_\tau_n \rightarrow \text{Bool} \) for \( n \geq 0 \) and some types \( \tau_1, \ldots, \tau_n \). A ground first-order quantifier-free formula (FO-formula) is a Boolean combination of predicates. We assume that set \( F \) contains the standard symbols of integer arithmetic along with uninterpreted functions, and their interpretation is standard. In particular, the sorts \( s_{\text{Bool}} \) and \( s_{\text{Int}} \) are the sorts of Booleans and integers, respectively. The sorts for the other types are uninterpreted. Hence, we deal with the formulas of logic QF_UFNIA [BFT17]. (Integer arithmetic in TLA⁺ does not have to be linear.)
3.5. Rewriting Framework

**Encoding Arenas in SMT** When rewriting a \( \text{KERA}^+ \) expression \( e \), our reduction system introduces new cells that encode symbolic values of \( e \)'s subexpressions. In SMT, these cells are introduced as constants of the respective sorts. To keep track of the arena edges, we introduce instrumental Boolean constants in SMT. Formally, given an arena \( \mathcal{A} = (\mathcal{V}, \mathcal{E}) \), for each edge \( e \in \mathcal{E} \), we introduce a Boolean constant \( \text{en}(e) \), whose value indicates, whether the edge \( e \) is enabled or not.

**Remark 3.5.2.** Consider the edge \( e_{41} = (c_4, 1, c_1) \) in Figure 3.2 (a). If \( \text{en}(e_{41}) \) evaluates to true, then the cell \( c_1 \) belongs to the set encoded by the set \( c_4 \); otherwise, \( c_1 \) does not belong to the set.

3.5.4 Abstract Reduction System (ARS)

We assume fixed a finite set of variables \( \text{Vars} \) that are used in \( \text{KERA}^+ \) expressions as free or bound variables. We define an abstract reduction system \((\mathcal{S}, \rightsquigarrow)\), where \( \mathcal{S} \) are the states of the reduction system and \( \rightsquigarrow \subseteq \mathcal{S} \times \mathcal{S} \) is a transition relation. A state of the abstract reduction system is defined as a tuple \((e, \mathcal{A}, \nu, \Phi)\), whose elements have the following meaning:

- \( e \) is a \( \text{KERA}^+ \) expression, possibly containing cells,
- \( \mathcal{A} \) is an arena,
- \( \nu \) is a partial function from \( \text{Vars} \) to \( \mathcal{V}(\mathcal{A}) \), which is called binding, and
- \( \Phi \) is a set of first-order formulas, which represents SMT constraints.

We define \( \rightsquigarrow \) via a set of reduction rules. For instance, the rules (\text{BOOL}) and (\text{INT}) below define transitions that reduce Boolean and integer literals to cells.

In the reduction rules, we write the premises above the bar and the expression with the current state and the new state of the reduction system below the bar. By convention, the state is written with the notation \( \langle e | \mathcal{A} | \nu | \Phi \rangle \).

\[
\begin{array}{c|c}
\text{Premise} & \text{b is FALSE or TRUE} \\
\text{Rule (BOOL)} & \langle b | \mathcal{A} | \nu | \Phi \rangle \rightsquigarrow \langle c | \mathcal{A}, c : \text{Bool} | \nu | \Phi, c = b \rangle \\
\text{Premise} & n \text{ is } 0, 1, -1, \ldots \\
\text{Rule (INT)} & \langle n | \mathcal{A} | \nu | \Phi \rangle \rightsquigarrow \langle c | \mathcal{A}, c : \text{Int} | \nu | \Phi, c = n \rangle \\
\text{Premise} & \triangleright \text{ is one of } <, \leq, >, \geq, =, \neq \\
\text{Rule (INTCMP)} & \langle c_l \triangleright c_r | \mathcal{A} | \nu | \Phi \rangle \rightsquigarrow \langle \text{res}_{\text{res}} | \mathcal{A}, \text{res}_{\text{res}} : \text{Bool} | \nu | \Phi, c_{\text{res}} \iff c_l \triangleright c_r \rangle
\end{array}
\]

Once we have introduced integer cells for the literals, we can reduce integer comparisons using the rule (\text{INTCMP}) and reduce integer arithmetics using the rule (\text{INTARITH}). The reduction rules add new SMT constraints to the set \( \Phi \).
3. TLA⁺ Model Checking Made Symbolic

<table>
<thead>
<tr>
<th>Premise</th>
<th>o is one of +, −, *, ÷, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule (IntArith)</td>
<td>( c_\ell \circ c_r</td>
</tr>
</tbody>
</table>

In general, expressions contain multiple operators and thus cannot be reduced with a single rule. The rule (RedArg) rewrites operator arguments from left to right. Unless stated otherwise, we assume that this rule can be freely applied to an expression before the other rules are applied. A few KerA⁺ operators require special treatment, e.g., \( \exists x \in S : p \) and \{x \in S : p\}.

**Premises**

| Rule (RedArg) | \( \langle e_i | A_{i-1} | \nu_{i-1} | \Phi_{i-1} \rangle \sim \langle c_i | A_i | \nu_i | \Phi_i \rangle \) for \( 1 \leq i \leq n \) |

To apply the reduction system to a KerA⁺ expression \( e \), e.g., to Init and Next, we introduce an initial state \( \langle e_0 | A_0 | \nu_0 | \Phi_0 \rangle \), whose arena, binding, and SMT constraints are empty. Formally, \( A_0 = (\emptyset, \emptyset) \), \( \Phi_0 = \emptyset \), and \( \nu(x) = \bot \) for \( x \in \text{Vars} \). Usually, the expression \( e_0 \) is a formula, that is, it has type \text{Bool}. For simplicity, we also assume that all constants that appear in \( e_0 \) have basic types, that is, \text{Int}, \text{Bool}, and \text{Name}, while the expressions of more complex types are constructed with built-in TLA⁺ operators. This restriction is not crucial, as one can initialize TLA⁺ parameters (called “CONSTANTS” in TLA⁺) by evaluating an additional formula, similar to Init. Then, we apply the reduction rules until one of the following states is reached: (1) an error state, in which no rule applies, or (2) a terminal state, in which the expression is a single cell. If an error state has been reached, then the expression \( e \) is not well-formed.

When a terminal state \( c_{\text{term}} \) is reached, and the terminal cell \( c_{\text{term}} \) has type \text{Bool}, we add the assertion \( c_{\text{term}} \) to the SMT constraints and check their satisfiability. In Sections 3.6-3.10 we introduce rewriting rules for sets, functions, tuples, records, sequences, and control operators. Section 4.2 contains soundness proofs.

3.6 Sets

Sets lie in the theoretical foundation of TLA⁺, as it builds upon Zermelo-Frānkël set theory with choice (ZFC). Hence, in theory, every TLA⁺ value is a set. However, in practice, we distinguish sets from the other objects, that is, Booleans, integers, functions, tuples, records, and sequences. One implication of using ZFC is that every set is constructed out of sets of smaller rank, the terminal sets being the objects of non-set types (or empty sets). Importantly, we only consider finite sets.

**Set Enumeration.** The simplest way to construct a set is by enumerating its elements, e.g., by writing \( \{1, 2, 3\} \). The rule (Enum) reduces a set of cells to a fresh cell \( c_{\text{set}} \). The rule links the elements \( c_1, \ldots, c_n \) to \( c_{\text{set}} \) in the arena and adds the constraint \( en(c_{\text{set}}, i, c_i) \) for each \( 1 \leq i \leq n \). Several important observations should be made. First, we only add constraints on the edges from \( c_{\text{set}} \) to the cells \( c_1, \ldots, c_n \), as the reduction rules for sets
refer only to the cells pointed by \(c_{set} \) in the arena. Second, the set elements may be not unique, as uniqueness test cannot be done at the time of rewriting, and most set operations do not require uniqueness. In other words, we encode multisets.

\[
\text{Rule (ENUM)} \quad \langle \{c_1, \ldots, c_n\} : \text{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi \rangle \\
\rightarrow \langle c_{set} : \text{Set}[\tau] \mid \mathcal{A}, c_{set}, c_{set} \rightarrow c_1, \ldots, c_n \mid \nu \mid \Phi, \land_{1 \leq i \leq n} \text{en}(c_{set}, i, c_i) \rangle
\]

**Set Membership.** An expression \(c_x \in c_S \) such that \(c_S \rightarrow \mathcal{A} c_1, \ldots, c_n \) is reduced to \(\forall_{1 \leq i \leq n} c_x = c_i \).

**Set Filter.** An expression \(\{x \in S : p\} \) constructs the set \(T\) that has only the elements of \(S\) that satisfy the predicate \(p\).

\[
\text{Premises} \quad \langle p[c_1/x], \ldots, p[c_n/x] \mid \mathcal{A} \mid \Phi \mid \nu \rangle \rightarrow \langle c^p_1, \ldots, c^p_n \mid \mathcal{A}' \mid \Phi' \mid \nu' \rangle
\]

\[
\text{Rule (FILTER)} \\
\rightarrow \langle \{x \in c_S : p\} : \text{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi \rangle \\
\rightarrow \langle c_T : \text{Set}[\tau] \mid \mathcal{A}', c_T \rightarrow c_1, \ldots, c_n \mid \nu' \mid \Phi', \text{InFilter} \rangle
\]

The rule (FILTER) implements this semantics in two steps. First, it reduces the applications of predicate \(p\) to all potential set elements \(c_1, \ldots, c_n\), that is, it rewrites the expressions \(p[c_i/x]\) for \(1 \leq i \leq n\). (As usual, the notation \(p[e/x]\) means that \(x\) is replaced by \(e\) in \(p\).) Second, it adds the constraint \(\text{InFilter}\) that requires every cell \(c_i\) to be in the new set \(c_T\) if and only if it is in \(c_S\) and it satisfies the predicate \(p\) instantiated to \(c_i\), that is, \(c_i^p\) is true:

\[
en(c_T, i, c_i) \Leftrightarrow (c_i^p \land \text{en}(c_S, i, c_i)) \text{ for } 1 \leq i \leq n \quad (\text{InFilter})
\]

**Union of Sets.** By definition, UNION \(S\) produces the set that comprises of the elements of the sets in \(S\). For example, UNION \(\{\{1, 2\}, \{2, 3\}\}\) produces the set \(\{1, 2, 3\}\). The rule (UNION) captures this. It introduces a fresh cell \(c_U\) for the union and points to the cells pointed by the descendants of \(c_S\).

\[
\text{Premises} \quad \langle c_S \rightarrow \mathcal{A} c^1_S, \ldots, c^n_S \rangle \\
\quad \langle c_S \rightarrow \mathcal{A} c^1_1, \ldots, c^1_{m_1} \rangle \text{ for } 1 \leq i \leq n
\]

\[
\text{Rule (UNION)} \\
\rightarrow \langle \text{UNION } c_S : \text{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi \rangle \\
\rightarrow \langle c_U : \text{Set}[\tau] \mid \mathcal{A}, c_U, c_U \rightarrow c^1_1, \ldots, c^1_{m_1}, c^2_1, \ldots, c^2_{m_2}, \ldots, c^n_{m_n} \mid \nu \mid \Phi, \text{InU}\rangle
\]

The SMT constraint \(\text{InU}\) simply requires a cell \(c^j_U\) to be in \(c_U\) if and only of it is in the set containing it, that is, in \(c^j_S\), and the set \(c^j_S\) belongs to \(c_U\):

\[
en(c_U, idx_{i,j}, c^j_U) \Leftrightarrow \left(\text{en}(c^j_S, j, c^j_U) \land \text{en}(c_S, i, c^j_S)\right) \text{ for } 1 \leq i \leq n, 1 \leq j \leq m_i, \quad (\text{InU})
\]

where the edge index \(idx_{i,j}\) is defined as \(m_1 + \cdots + m_{i-1} + j\).
Figure 3.3: An arena constructed for the set comprehension \( \{ x \div 3 : x \in \{ 2, 3, 4 \} \} \). Every cell \( c_i \) has value \( i \) for \( 0 \leq i \leq 4 \). Cell \( c_S \) encodes the set \( \{ 2, 3, 4 \} \), and cell \( c_T \) encodes the result of the set comprehension.

The constraint \( (\text{In}U) \) may seem to be unsound. Indeed, consider the arena in Figure 3.2(a) and assume that we compute \( c_5 \). Further, assume that the SMT solver sets \( \text{en}(c_5, 1, c_2) \) to true and \( \text{en}(c_2, 1, c_2) \) to false, that is, 2 is a member of the set encoded by \( c_5 \) and 2 is not a member of the set encoded by \( c_4 \). Equation \( (\text{In}U) \) produces the following constraints (among others): \( \text{en}(c_U, 2, c_2) \leftrightarrow \text{en}(c_4, 2, c_2) \land \text{en}(c_6, 1, c_4) \) and \( \text{en}(c_U, 3, c_2) \leftrightarrow \text{en}(c_5, 1, c_2) \land \text{en}(c_6, 2, c_5) \). As a result, \( \text{en}(c_U, 2, c_2) \) is false, whereas \( \text{en}(c_U, 3, c_2) \) is true. There is no contradiction here, as for the set membership of \( c_2 \) in \( c_U \), it is sufficient to find one enabled edge, that is, \( (c_U, 3, c_2) \).

**Set Map.** By definition, \( \{ e : x \in S \} \) constructs the set \( T \) with the following property: For every \( x \), it holds that \( z \in T \) if and only if there is \( y \in S \) such that \( z = e[y/x] \). For example, the expression \( \{ x \div 3 : x \in \{ 2, 3, 4 \} \} \) constructs the set \( \{0, 1\} \). The operator \( \div \) denotes integer division in TLA\(^+\). Rule (MAP) implements this. Figure 3.3 shows the arena that is constructed in the process of reduction.

<table>
<thead>
<tr>
<th>Premises</th>
<th>Rule (MAP)</th>
</tr>
</thead>
</table>
| \( c_S \rightarrow \text{A} c_1, \ldots, c_n \)  
\( \langle e[c_1/x], \ldots, e[c_n/x] \mid \text{A} \mid \Phi \mid \nu \rangle \rightarrow \langle c^*_1 : \tau, \ldots, c^n : \tau \mid \text{A}' \mid \Phi' \mid \nu' \rangle \) | \( \langle e : x \in c_S \rangle \mid \text{A} \mid \nu \mid \Phi \)  
\( \langle c_T \mid \text{A}' \rangle, c_T : \text{Set}[\tau], c_T : c^*_1, \ldots, c^n : \nu' \mid \Phi', \text{InMap} \rangle \) |

The rule works in two steps. First, it reduces the applications of expression \( e \) to all potential set elements \( c_1, \ldots, c_n \), that is, it rewrites the expressions \( e[c_i/x] \) to \( c^*_i \) for \( 1 \leq i \leq n \). Second, the constraint \( (\text{InMap}) \) enforces that a cell \( c^*_i \) belongs to the set encoded by the cell \( c_T \) if and only if its preimage \( c_i \) belongs to the set encoded by the cell \( c_S \): \( \text{en}(c_T, i, c^*_i) \Leftrightarrow \text{en}(c_S, i, c_i) \) for \( 1 \leq i \leq n \) \( (\text{InMap}) \).

**Remark 3.6.1.** Consider Figure 3.3. The cell \( c_1 \) is mapped to the cell \( c_0 \), whereas the cells \( c_3 \) and \( c_4 \) are mapped to the cell \( c_1 \). Assume that the SMT solver sets \( \text{en}(c_S, 3, c_4) \) to true and \( \text{en}(c_S, 2, c_3) \) to false. Hence, \( \text{en}(c_T, 3, c_1) \) holds true and \( \text{en}(c_T, 2, c_1) \) does not. Still, \( c_1 \) belongs to the set encoded by \( c_T \), as the edge \( (c_T, 3, c_1) \) is enabled. \(<

**Integer Interval** \( a..b \). This operator is quite often used in TLA\(^+\) to define the set \( \{ i \in \mathbb{Z} : a \leq i \leq b \} \). The latter set cannot be defined in KER\(^+\), as our language supports only finite sets. When the bounds \( a \) and \( b \) are integer constants, we reduce \( a..b \) to the
set enumeration \( \{a, a+1, \ldots, b\} \). Otherwise, the user has to find a static set \( S \supseteq a \ldots b \) that can be filtered by the \( \text{KerA}^+ \) expression \( \{i \in S: a \leq i \wedge i \leq b\} \). It is often easy to find such a set \( S \), as the specification parameters are fixed.

### Set Equality

As sets are encoded as constants of uninterpreted sorts in SMT, it is not sound to use the SMT equality. One way of imposing equality constraints is by writing down the set equality axioms as done by [MV18]. However, such axioms immediately introduce quantified formulas in SMT. Instead of axioms, we implement lazy equality in the rule \( \text{(SetEq)} \). Whenever two cells \( c_S \) and \( c_T \) are compared for the first time, \( \text{(SetEq)} \) rewrites the definition of set equality into a Boolean cell \( c_{eq} \). Additionally, it adds the SMT constraint \( c_S = c_T \iff c_{eq} \), which allows us to use SMT equality in the later occurrences of \( c_S = c_T \).

#### Premise
\[
\langle (\forall x \in c_S : x \in c_T) \land (\forall x \in c_T : x \in c_S) \mid A \mid \nu \mid \Phi \rangle
\]

#### Rule (SetEq)
\[
\langle c_{eq} : \text{Bool} \mid A' \mid \nu' \mid \Phi' \rangle
\]

### Set Cardinality

In TLA\(^+\), an expression \( \text{Cardinality}(S) \) produces a natural number that equals to the number of elements in a finite set \( S \). Cardinalities are used in TLA\(^+\) specifications in various ways. For instance, to compare cardinalities, that is, \( \text{Cardinality}(S) \geq \text{Cardinality}(T)/2 + 1 \), or to construct a set of integers \( 1 \ldots \text{Cardinality}(S) \), or as a function argument. Hence, we use a generic approach to computing the set cardinality by the recurrence relation in Equation (3.1), assuming that a set cell \( c_S \) is pointing to the element cells \( c_1, \ldots, c_n \):

\[
k_0 = 0 \quad \text{and} \quad k_{i+1} = \text{ITE}(en(c_S, i, c_i) \land \text{notSeen}_i, 1 + k_i, k_i) \text{ for } 0 < i \leq n \quad (3.1)
\]

Equation (3.2) requires that the \( i \)th element contributes to the cardinality, if the previously considered elements are either outside of the set, or are different from the \( i \)th element:

\[
\text{notSeen}_i = \bigwedge_{1 \leq j < i} (en(c_S, j, c_j) \rightarrow c_j \neq c_i) \text{ for } 0 < i \leq n \quad (3.2)
\]

Hence, \( \text{Cardinality}(c_S) = k_n \). A more efficient approach can be applied to a more restricted fragment, e.g., BAPA by [KNR05]. We plan to use specialized approaches in the future.

### 3.7 Picking Set Elements

While developing rewriting rules for TLA\(^+\) operators, we found that many rules can be reduced to the auxiliary operator \( \text{FROM} e_1, \ldots, e_n \) by \( \theta \), where \( \theta \) is an integer constant and \( e_1, \ldots, e_n \) are TLA\(^+\) expressions of the same type \( \tau \). The meaning of this operator
is as follows: If $\theta \in 1..n$, then FROM $e_1, \ldots, e_n$ BY $\theta$ returns $e_\theta$; Otherwise, it returns an arbitrary value of type $\tau$. The constant $\theta$ defines the value to be picked from the sequence $e_1, \ldots, e_n$. Hence, we call it an oracle.

The operator FROM $e_1, \ldots, e_n$ BY $\theta$ is not part of TLA+. The syntax for TLA+ proofs has a similar operator $\text{pick } x \in S$, which returns an arbitrary element of the set $S$ \cite{lamport18}.

However, $\text{pick}$ does not provide us with fine control of which element could be picked. We define several reduction rules for FROM $e_1, \ldots, e_n$ BY $\theta$, which vary by the types of the expressions $e_1, \ldots, e_n$.

### Picking Basic Values

The rule (FROMBASIC) applies to Booleans, integers, and constants. It introduces a new cell $c_{\text{pick}}$ and requires that $c_{\text{pick}}$ equals to the $\theta$th value as prescribed by the oracle. When the oracle has a value outside of $1..n$, the picked value is unconstrained.

**Premise**: $c_1 : \tau, \ldots, c_n : \tau$, $\tau$ is basic

**Rule (FROMBASIC)**

$$\frac{ \langle \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \mid \mathcal{A} \mid \nu \mid \Phi \rangle }{ \Rightarrow \langle c_{\text{pick}} \mid \mathcal{A}, c_{\text{pick}} : \tau \mid \nu \mid \Phi, \land_{1 \leq i \leq n} (\theta = i \Rightarrow c_{\text{pick}} = c_i) \rangle }$$

### Picking Sets

The second rule (FROMSET) picks a set element which is itself a set. This is the most intricate rule, as it requires us to construct a set that mimics the structure of every set that is captured by the cells $c_1, \ldots, c_n$. The rule assumes that every cell $c_i$ has the same type $\text{Set}[\tau]$ for some type $\tau$ and $1 \leq i \leq n$. Without loss of generality, we assume that every cell points to exactly the same number of cells, that is, if $c_i \rightarrow A c_1^i, \ldots, c_k^i$ and $c_j \rightarrow A c_1^j, \ldots, c_m^j$, then $k = m$. If it is not the case we can introduce additional edges by replicating the last element of the sequence, e.g., if $k < m$, then we would extend the arena as $c_i \rightarrow A c_1^i, \ldots, c_k^i, \ldots, c_k^i$, where $c_k^i$ is repeated $m - k + 1$ times. (When $k = 0$, we copy the elements from the longest sequence and disable the new edges.)

The rule (FROMSET) works in two steps. First, for every index $j \in 1..m$, it picks an element $c_j^\text{pick}$ among the $j$th elements of the sets $c_1, \ldots, c_n$. Importantly, the opera-
3.8. Tuples and Records

c_\ell : \text{Int} * \text{Bool} * \text{Name} \quad c_r : [a \mapsto \text{Int}, b \mapsto \text{Bool}, c \mapsto \text{Name}]

\begin{align*}
c_1 : \text{Int} & \quad c_2 : \text{Bool} & \quad c_3 : \text{Name} \\
\downarrow & \quad 2 & \quad 3
\end{align*}

Case (a)

\begin{align*}
c_4 : \text{Int} & \quad c_5 : \text{Bool} & \quad c_6 : \text{Name} \\
\downarrow & \quad 2 & \quad 3
\end{align*}

Case (b)

Figure 3.5: (a) The arena constructed for the tuple $(1, \text{TRUE}, "abc")$, assuming that the expressions $1$, $\text{TRUE}$, and "$abc"$ were rewritten into cells $c_1$, $c_2$, and $c_3$. (b) The arena constructed for the record $[a \mapsto 1, b \mapsto \text{TRUE}, c \mapsto "abc"]$, assuming that the expressions $1$, $\text{TRUE}$, and "$abc"$ were rewritten into cells $c_4$, $c_5$, and $c_6$.

tors FROM $c_1^j, \ldots, c_n^j$ BY $\theta$ are using the same oracle $\theta$ for every $j \in 1..m$. As a result, they pick the respective elements from the same set $c_\theta$. Second, the resulting set $c_{\text{res}}$ points to the picked elements $c_1^{\text{pick}}, \ldots, c_m^{\text{pick}}$.

Premise

$\begin{align*}
&c_i : \text{Set}[\tau] \quad \text{for } 1 \leq i \leq n \\
&c_i \rightarrow \text{A}_0, c_1^i, \ldots, c_n^i \quad \text{for } 1 \leq i \leq n \\
&\theta : \text{Int} \\
&\langle \text{FROM} c_1^j, \ldots, c_n^j \text{ BY } \theta \mid \text{A}_{j-1} \mid \nu_{j-1} \mid \Phi_{j-1} \rangle \\
&\leadsto \langle c_{\text{pick}}^j : \tau \mid \text{A}_j \mid \Phi_j \mid \nu_j \rangle \text{ for } 1 \leq j \leq m
\end{align*}$

Rule (FROMSET)

$\begin{align*}
\langle \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \mid \text{A}_0 \mid \nu_0 \mid \Phi_0 \rangle \\
\leadsto \langle c_{\text{res}} \mid \text{A}, c_{\text{res}} : \text{Set}[\tau], c_{\text{res}} \rightarrow c_{\text{pick}}^1, \ldots, c_{\text{pick}}^m \mid \nu_m \mid \Phi_m, \text{InPicked} \rangle
\end{align*}$

The constraint \text{InPicked} requires the new set cell $c_{\text{res}}$ to contain a cell $c_{\text{pick}}^j$ if and only if the respective set chosen by the oracle $\theta$ contains the $j$th cell.

$en(c_{\text{res}}, j, c_{\text{pick}}^j) \iff \bigvee_{1 \leq i \leq n} \theta = i \land en(c_i, j, c_i^j) \text{ for } 1 \leq j \leq m$ \hspace{1cm} (InPicked)

Remark 3.7.1. Figure 3.4 shows an example of the rule applied to \text{FROM} $c_1, c_2$ \text{BY} $\theta$. The cells $c_1$ and $c_2$ have type $\text{Set}[\tau]$, each of them pointing to three element cells $c_1^1, c_1^2$, and $c_1^3$ for $i \in \{1, 2\}$. The rule first applies \text{FROM} $c_1^j, c_2^j$ \text{BY} $\theta$ three times for $j \in \{1, 2, 3\}$ to pick one element $c_{\text{pick}}^j$ from each pair. Note that use of $\theta$ guarantees us that the elements are drawn from the same set. The resulting cell $c_{\text{res}}$ is pointing to the three picked cells $c_1^{\text{pick}}, c_2^{\text{pick}}$, and $c_3^{\text{pick}}$.

Picking Other Values We have also defined the rules for picking a value from: a set of functions, a set of tuples, a set of records, a set of sequences, and a powerset (constructed with \text{SUBSET} $S$). They are similar to (FROM\text{BASIC}) and (FROM\text{SET}) and are omitted for brevity.
3.8 Tuples and Records

Tuples and records are easy to express in our framework, since the types give us precise information about the number of fields and their types. Importantly, we assume that the tuple elements and record fields are accessed with constant expressions, e.g., `tuple[3]` or `record.name`, but not `tuple[x]` and `record[x]`, where `x` is a variable. This is usually the case for TLA+ specifications.

**Tuple Constructor.** A tuple constructor adds a new cell pointing to the element cells in their index order. Figure 3.5 (a) shows an example of applying the rule (TupCTOR).

\[
\text{Rule (TupCTOR)} \quad \langle c_1, \ldots, c_n : \tau_1 \cdots \tau_n \mid A \mid \nu \mid \Phi \rangle \\
\leadsto \langle c_{\text{new}} : A, c_{\text{new}} : \tau_1 \cdots \tau_n, c_{\text{new}} \rightarrow c_1, \ldots, c_n \mid \nu \mid \Phi \rangle
\]

**Tuple Application.** The tuple application rule returns the `i`th cell pointed by the tuple cell:

\[
\text{Rule (TupApp)} \quad c_t \rightarrow_i A \quad c_1, \ldots, c_n \quad i \in \{1, \ldots, n\}
\]

\[
\langle c_t[i] \mid A \mid \nu \mid \Phi \rangle \leadsto \langle c_i \mid A \mid \nu \mid \Phi \rangle
\]

**Tuple Domain.** For a tuple `t` of type `\tau_1 \cdots \tau_n`, the expression `\text{DOMAIN } t` is reduced to `1..n`.

**Records.** The rules for records are similar to the rules for tuples. We assume that the field names in each record type `[nm_1 \mapsto e_1, \ldots, nm_n \mapsto e_n]` are lexicographically sorted. Obviously, there is bijection between `{nm_1, \ldots, nm_n}` and `1..n`. Hence, we use the rules for tuples to rewrite most of the record operators. The only exception is `\text{DOMAIN } r`, which returns the set `{nm_1, \ldots, nm_n}`. Figure 3.5 (b) shows an example of rewriting a record constructor.

3.9 Functions and Sequences

Functions are the second most used data structure after sets in TLA+. [Lam02] introduces tuples, sequences, and records as functions, so in pure TLA+ any data structure different from a set is a function. As KERA+ is well-typed, we treat general functions differently from tuples, records, and sequences. A function in KERA+ has a type `\tau_1 \rightarrow \tau_2`, which implies that it always returns elements of the same type. Below, we define the reduction rules for function operators. In arenas, we encode a function `f` with its associated relation, that is, as the set of pairs `{(x, f[x]) : x \in \text{DOMAIN } f}`. As a result, we reuse the rules for sets (Section 3.6) and tuples (Section 3.8). For instance, equality of two functions is simply the set equality of their associated relations.
At the arena level, a function cell \( c_f \) is always pointing to a single cell that stores the associated relation. See Figure 3.2 (c) for example. We use the notation \( \text{funrel}(c_f) \) to refer to this relation cell.

**Function Definition (FunCTOR).** In TLA\(^+\), an expression \([x \in S \mapsto e] \) defines a function with the domain \( S \) that maps every value \( v \in S \) to \( e[v/x] \), where \( x \) is substituted with \( v \) in the expression \( e \) (see [Lam02] p. 302). This expression is similar to the set map \( \{ e : x \in S \} \). Hence, for the function constructor \([x \in S \mapsto e] \), we apply the rewriting rule (SetMap) to the expression \( \{(x, e) : x \in S\} \). This rule produces a cell \( c_{\text{rel}} \) that encodes the associated relation \( c_{\text{rel}} \) of type \( \text{Set}[\tau_1 \times \tau_2] \), where \( \tau_1 \) is the type of elements of \( S \), and \( \tau_2 \) is the type of \( e \). We add a cell \( c_f \) of type \( \tau_1 \mapsto \tau_2 \) and make it point to \( c_{\text{rel}} \), that is, \( c_f \mapsto_A c_{\text{rel}} \). The rule (FunCTOR) produces \( c_f \) as a result.

**Function Domain (FunDOM).** Assuming that \( f \) is reduced to a cell \( c_f \), we rewrite \( \text{DOMAIN} \) \( c_f \) as \( \{t[1] : t \in \text{funrel}(c_f)\} \), that is, we map every pair in the relation \( \text{funrel}(c_f) \) to its first element.

**Function Update (FunExc).** In TLA\(^+\), an expression \([f \text{ EXCEPT } ![a] = r]\) produces a new function \( g \) that has three properties: (1) It has the same domain as \( f \), (2) \( g[x] = f[x] \) for \( x \in \text{DOMAIN} \ f \setminus \{a\} \), and (3) \( g[a] = r \) if \( a \in \text{DOMAIN} \ f \). (See [Lam02] p. 302.) Assuming that expression \( f \) has been rewritten to a cell \( c_f \), we update the associated relation \( \text{funrel}(c_f) \) as follows:

\[
\{\text{ITE}(p[1] = a, \langle a, r \rangle, p) : p \in \text{funrel}(c_f)\} \quad (\text{Except})
\]

In (Except), all pairs that contain \( a \) as the first component are replaced with the pair \( \langle a, r \rangle \), while the other pairs stay unchanged. It is easy to see that the above properties (1)-(3) are satisfied. We give the rewriting rule for ITE in Section 3.10.

**Function Application (FunApp).** In TLA\(^+\), an expression \( f[e] \) returns the result of applying the function \( f \) to \( e \), provided that \( e \in \text{DOMAIN} \ f \). When \( e \notin \text{DOMAIN} \ f \), the result is unspecified. The rule (FunAPP) implements this semantics.

| Premises | \( c_{\text{fun}} \overset{1}{\rightarrow_A} c_{\text{rel}} \overset{\text{FROM} \ c_1, \ldots, c_n}{\rightarrow_A} \cdots \overset{\text{BY} \ c_{\text{ora}}}{\rightarrow} \text{Int} \ | \nu \ | \Phi, 0 \leq c_{\text{ora}} \leq n \) |
|----------|----------------------------------------------------------------------------------------------------------------------------------|
| Rule (FunAPP) | \( c_{\text{fun}}[c_{\text{arg}}] \rightarrow_A \nu \ | \Phi \rightarrow \nu_2 | \Phi_2 | \begin{cases} \text{WhenInDomain} & \text{WhenOutsideDomain} \end{cases} \) |

First, the rule (FunAPP) introduces an integer oracle \( c_{\text{ora}} \), which points either to a cell from \( c_1, \ldots, c_n \) (when \( 1 \leq c_{\text{ora}} \leq n \)), or an arbitrary cell of proper type (when \( c_{\text{ora}} = 0 \)). Second, a cell \( c_{\text{pair}} \) is picked using the operator \( \text{FROM} \ c_1, \ldots, c_n \). This is the tuple that comprises a function argument and the respective result, so the rule (FunAPP)
returns the function result \( c_{para}[2] \). Third, the SMT formula (WhenInDomain) requires the oracle to pick the right pair, that is, the one that actually belongs to the relation and whose first component is equal to the argument. Finally, the SMT formula (WhenOutsideDomain) allows the oracle value to be zero, only if there is no pair that matches the passed argument \( c_{arg} \). Importantly, as the rule uses equality, we require that the lazy equality constraints \( c_{arg} = c_i[1] \) are generated for \( 1 \leq i \leq n \).

\[
\begin{align*}
    c_{ora} = i &\Rightarrow (c_i[1] = c_{arg} \land \text{en}(c_{fun}, i, c_i)) \text{ for } 1 \leq i \leq n \quad \text{(WhenInDomain)} \\
    c_{ora} = 0 &\Rightarrow (c_i[1] \neq c_{arg} \lor \neg\text{en}(c_{fun}, i, c_i)) \text{ for } 1 \leq i \leq n \quad \text{(WhenOutsideDomain)}
\end{align*}
\]

**Sequences.** We briefly discuss sequences. In principle, sequence operators can be expressed with function operators, we omit them here for brevity. However, these equivalent expressions are unnecessarily complex. Instead, we encode a sequence \( q \) of type \( \text{Seq}[\tau] \) as a tuple \( \langle \text{start, end, fun} \rangle \). The components \text{start} and \text{end} are integers that store the first index of the sequence and the index right after the end of the sequence respectively. The component \text{fun} is a function of type \( \text{Int} \to \tau \) that maps integers \( 1..n \) to values of type \( \tau \) for some \( n \geq 0 \). The sequence operators maintain the invariant: \( \text{start} \geq 1 \land \text{end} \leq n + 1 \). Hence, the elements of sequence \( q \) are in the window of indices \( [\text{start, end}] \).

### 3.10 Control Operators and Quantifiers

**Branching.** The operator \( \text{ite}(c_p, c_1, c_2) \) returns the value of one of its branches, depending on the Boolean condition \( c_p \). We use the \( \text{from}(c_1, c_2) \) for \( \theta \in \{1, 2\} \).

<table>
<thead>
<tr>
<th>Premise</th>
<th>( \langle \text{from}(c_1, c_2) \rangle \cdot \langle A, \theta : \text{Int} \mid \nu \mid \Phi, 1 \leq \theta \leq 2 \rangle \leadsto (c_{res} \mid A_2 \mid \nu_2 \mid \Phi_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule ( \text{ITE} )</td>
<td>( \langle \text{ite}(c_p, c_1, c_2) \rangle : \tau \mid A \mid \nu \mid \Phi \rangle \leadsto (c_{res} \mid A_2 \mid \nu_2 \mid \Phi_2, \theta = 1 \Leftrightarrow c_p) )</td>
</tr>
</tbody>
</table>

Interestingly, we do not compare \( c_{res} \) to \( c_1 \) and \( c_2 \), as one would expect from the standard if-then-else semantics. Instead, we delegate the job to the oracle \( \theta \).

**Assignments.** An assignment \( x' \in c_S \) in \( \text{KERA}^+ \) specifies that a variable \( x' \) takes a value from the set \( S \). Since any element of the set may be chosen, we use \( \text{from}(c_1, \ldots, c_n) \) for the cells pointed by the cell \( c_S \). We reserve the value \( \theta = 0 \) for the case when the set is empty, which results in assigning an arbitrary value of proper type to the variable \( x' \).

<table>
<thead>
<tr>
<th>Premises</th>
<th>( \langle x' \in c_S \rangle \mid A \mid \nu \mid \Phi )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \langle \text{from}(c_1, \ldots, c_n) \rangle \cdot \langle A, \theta : \text{Int} \mid \nu \mid \Phi, 0 \leq \theta \leq n \rangle )</td>
</tr>
<tr>
<td></td>
<td>( \leadsto (c \mid A_2 \mid \nu_2 \mid \Phi_2) )</td>
</tr>
<tr>
<td>Rule ( \text{ASGN} )</td>
<td>( \langle x' \in c_S \rangle \mid A \mid \nu \mid \Phi )</td>
</tr>
<tr>
<td></td>
<td>( \leadsto (\text{true} \mid A_2 \mid \nu_2[x \mapsto c] \mid \Phi_2, \theta = 0 \Leftrightarrow \bigwedge_{1 \leq i \leq n} \neg\text{en}(c_S, i, c_i), )</td>
</tr>
<tr>
<td></td>
<td>( \bigwedge_{1 \leq i \leq n} (\theta \neq i \lor \neg\text{en}(c_S, i, c_i)) \rangle )</td>
</tr>
</tbody>
</table>

We omit the rules for the assignments \( f' \in \text{subset} S \) and \( f' \in [S \to T] \) for brevity.
Substitution. A variable \( x \) can be replaced with the cell given by a valuation \( \nu \):

\[
\begin{array}{c|c}
\text{Premise} & x \in \text{Vars} \\
\text{Rule (SUB)} & \langle x \mid \mathcal{A} \nu \mid \Phi \rangle \rightsquigarrow \langle \nu(x) \mid \mathcal{A} \nu \mid \Phi \rangle
\end{array}
\]

Existential Quantifiers. Quantified expressions are a fundamental building block of TLA\(^+\), as well as KERA\(^+\). Since we consider only finite sets, an existential quantifier can be replaced with disjunction. If the body of the quantified expression contains variable assignments, we translate \( \exists x \in \mathcal{C}_S : \mathcal{P} \) as the non-deterministic disjunction \( \mathcal{P}[c_1/x] \oplus \ldots \oplus \mathcal{P}[c_n/x] \), where \( \mathcal{C}_S \) is pointing to \( c_1, \ldots, c_n \).

\[
\begin{array}{c|c}
\text{Premise} & \mathcal{C}_S \rightarrow_{\mathcal{A}} c_1, \ldots, c_n \\
\text{Rule (EXISTS)} & \langle \exists x \in \mathcal{C}_S : \mathcal{P} \mid \mathcal{A} \nu \mid \Phi \rangle \rightsquigarrow \langle \mathcal{P}[c_1/x] \oplus \ldots \oplus \mathcal{P}[c_n/x] \mid \mathcal{A} \nu \mid \Phi \rangle
\end{array}
\]

Replacing an existential quantifier with a disjunction may seem to be suboptimal. However, we cannot avoid it, as existential quantification may be used to express universal quantification, e.g., \( \neg \exists \mathcal{V} \). In this case, we have to explore all possible valuations for \( \mathcal{V} \). In the implementation, we introduce the following optimization for existential quantifiers. We transform the formula such as Next into its negated normal form and check whether \( \exists \mathcal{V} \) is located under a universal quantifier. If this is not the case, we introduce a Skolem constant \( c \in \mathcal{C}_S \) and produce the expression \( \mathcal{P}[c/x] \) instead of the disjunction. As expected, this optimization significantly reduces the number of SMT constraints.

Operator choose. By definition, \( \text{choose} \mathcal{X} \in \mathcal{S} : \mathcal{P} \) returns an element of \( \mathcal{S} \) that satisfies the expression \( \mathcal{P} \) (see [Lam02, p. 294]). If there is no such an element, the result is undefined. Importantly, \( \text{choose} \) is deterministic: Two expressions \( \text{choose} \mathcal{X} \in \mathcal{S} : \mathcal{P} \) and \( \text{choose} \mathcal{Y} \in T : \mathcal{Q} \) have equal values, if the filtered sets are equal, that is, \( \{ \mathcal{X} \in \mathcal{S} : \mathcal{P} \} = \{ \mathcal{Y} \in T : \mathcal{Q} \} \).

\[
\begin{array}{c|c|c}
\text{Premise} & \langle \{ \mathcal{X} \in \mathcal{S} : \mathcal{P} \} \mid \mathcal{A} \nu \mid \Phi \rangle \rightsquigarrow \langle \mathcal{C}_\mathcal{X} : \tau \mid \mathcal{A}_2 \nu_2 \mid \Phi_2 \rangle \\
& \mathcal{C}_\mathcal{X} \rightarrow_{\mathcal{A}_2} c_1, \ldots, c_n \\
& \langle \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \mid \mathcal{A}_2, \theta : \text{Int} \mid \nu_2 \mid \Phi_2, 0 \leq \theta \leq n \rangle \\
& \rightsquigarrow \langle c_{\text{res}} \mid \mathcal{A}_3 \nu_3 \mid \Phi_3 \rangle \\
\text{Rule (CHOOSE)} & \langle \text{choose} \mathcal{X} \in \mathcal{S} : \mathcal{P} \mid \mathcal{A} \nu \mid \Phi \rangle \\
& \rightsquigarrow \langle c_{\text{res}} \mid \mathcal{A}_3 \nu_3 \mid \Phi_3, \text{choose}_\tau(\mathcal{C}_\mathcal{P}) = c_{\text{res}} \rangle
\end{array}
\]

The rule \( \text{CHOOSE} \) implements this semantics as follows. First, it rewrites the set \( \{ \mathcal{X} \in \mathcal{S} : \mathcal{P} \} \) into a cell \( \mathcal{C}_\mathcal{X} \) of some type \( \tau \). Suppose that \( \mathcal{C}_\mathcal{X} \) points to the element cells \( c_1, \ldots, c_n \). Second, the rule applies \( \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \) to pick a cell \( c_{\text{res}} \) using an oracle \( \theta \). The cell \( c_{\text{res}} \) is the result of rewriting the expression \( \text{CHOOSE} \mathcal{X} \in \mathcal{S} : \mathcal{P} \). To guarantee determinism of \( \text{CHOOSE} \), for each type \( \tau \), we introduce an uninterpreted function \( \text{choose}_\tau \) of sort \( \text{Set}[\tau] \rightarrow \tau \), and require \( \text{choose}_\tau(\mathcal{C}_\mathcal{P}) = c_{\text{res}} \). Finally, the rewriting system instantiates...
lazy equality between the pairs cells $c^1_F$ and $c^2_F$, as well as the pairs $\text{choose}_r(c^1_F)$ and
$\text{choose}_r(c^2_F)$, which are produced by rewriting of $\{x \in S : p\}$ and $\{y \in T : q\}$ in the rule
$\text{CHOOSE}$. Congruence of uninterpreted functions gives us the required determinism.

Non-deterministic Disjunction. This operator is used to combine symbolic transitions
$T_1, \ldots, T_k$. In contrast to the disjunction $\lor$, the operands of $\oplus$ produce independent
variable valuations. For the sake of presentation, we introduce the rule for the binary
case $A \oplus B$ and one variable $x'$. It is easy, though tedious, to extend this rule to multiple
variables and $n$-ary disjunctions.

\[
\begin{align*}
\text{Premise} & \quad \langle e_i \mid A_{i-1} \mid \nu_0 \mid \Phi_{i-1} \rangle \leadsto \langle c_i \mid A_i \mid \nu_i \mid \Phi_i \rangle \quad i = 1, 2 \\
& \quad \langle \text{from } \nu_1(x'), \nu_2(x') \text{ by } \theta \mid A_2, \theta : \text{Int } \mid \nu_0 \mid \Phi_2, \theta \in \{1, 2\} \\
& \quad \leadsto \langle c_2 \mid \nu_0 \mid \Phi_3 \rangle \\
\text{Rule (NDC)} & \quad \langle e_1 \oplus e_2 \mid A_0 \mid \nu_0 \mid \Phi_0 \rangle \\
& \leadsto \langle c_r \mid A_3, c_r : \text{Bool } \mid \nu_0 \circ [x' \mapsto c_x] \mid \Phi, c_r \Leftarrow c_1 \lor c_2, \theta = 1 \Rightarrow c_1, \theta = 2 \Rightarrow c_2 \rangle
\end{align*}
\]

3.11 Additional Reduction Rules from TLA$^+$ to KerA$^+$

In this section, additional reduction rules from TLA$^+$ to KerA$^+$ are provided.

Set membership. Two rules for membership test are missing in Chapter 3 Testing, whether a set $T$ belongs to the powerset of $S$ is easy: $T$ has to be a subset of $S$.

\[
\text{Rule (INPOWERSET) } \quad \langle T \in \text{subset } S \mid A \mid \nu \mid \Phi \rangle \leadsto \langle T \subseteq S \mid A \mid \nu \mid \Phi \rangle
\]

Testing, whether a function belongs to a set of functions defined via $[S \to T]$ boils down to testing whether the function domain and the image are subsets of $S$ and $T$ respectively.

\[
\begin{align*}
\text{Premise} & \quad c_f \to \lambda_A c_{rel} \\
\text{Rule (INFUNSET)} & \quad \langle c_f \in [S \to T] \mid A \mid \nu \mid \Phi \rangle \\
& \leadsto \langle \{t[1] : t \in c_{rel}\} \subseteq S \land \{t[2] : t \in c_{rel}\} \subseteq T \mid A \mid \nu \mid \Phi \rangle
\end{align*}
\]

Assignments. There are two special forms of assignments. The first form assigns to $x'$
a value from a powerset of $c_S$. To do so, we introduce a new cell $c_T$ that points to exactly
the same cells as $c_S$, but not all of these cells have to belong to $c_T$.

\[
\begin{align*}
\text{Premise} & \quad c_S \to_A c_1, \ldots, c_n \quad c_S : \text{Set}[\tau] \\
\text{Rule (ASSIGNFROMPOWERSET)} & \quad \langle x' \in \text{subset } c_S \mid A \mid \nu \mid \Phi \rangle \\
& \leadsto \langle \text{true } \mid A, c_T : \text{Set}[\tau], c_T \to_A c_1, \ldots, c_n \mid \nu \circ [x' \mapsto c_T] \mid \\
& \quad \Phi \text{InPowerset} \rangle
\end{align*}
\]

As some of the cells may be missing from $c_S$, we add a constraint that only those cells
that are actually in $c_S$ should be in $c_T$. 58
3.12 Implementation

The second form assigns to \( x' \) a function from a set of functions \([X \rightarrow Y]\). To do so, we introduce a cell to store the function relation \( \mathcal{c}_{rel} \) that contains the pairs from the sets \( X \) and \( Y \).

\[
\bigwedge_{i=1}^{n} \neg \text{in}(\mathcal{c}_i, \mathcal{c}_S) \Rightarrow \neg \text{in}(\mathcal{c}_1, \mathcal{c}) \quad \text{(InPowerset)}
\]

The final arena \( \mathcal{A}' \) is defined in the equation below. For convenience, we use the tuple notation \( (\mathcal{c}_i, \mathcal{c}_r^i) \) to introduce pairs. If we want to be precise, we have to introduce additional reduction steps that rewrite these pairs into cells.

\[
\mathcal{A}' = \mathcal{A}_m, \mathcal{c}_f : \tau_1 \rightarrow \tau_2, \mathcal{c}_{rel} : \text{Set}[\tau_1 \ast \tau_2], c_f \rightarrow \mathcal{A}' \mathcal{c}_{rel}, \mathcal{c}_{rel} \rightarrow \mathcal{A}' \langle \mathcal{c}_1, \mathcal{c}_r^1 \rangle, \ldots, \langle \mathcal{c}_m, \mathcal{c}_r^m \rangle \quad (3.3)
\]

To make sure that the new relation is using only the cells that actually belong to \( \mathcal{c}_S \), we introduce the constraint \( \text{(InDomain)} \).

\[
\text{en}(\mathcal{c}_{rel}, i, (\mathcal{c}_i, \mathcal{c}_r^i)) \Leftrightarrow \text{en}(\mathcal{c}_X, i, \mathcal{c}_i) \text{ for } 1 \leq i \leq n \quad \text{(InDomain)}
\]

Finally, to guarantee that \( \mathcal{c}_{rel} \) defines a function, we have to make sure that no two pairs disagree on the second component, if their first components are equal. This can happen, as the set \( \mathcal{c}_S \) may contain multiple copies of different but equal cells. To this end, we introduce an uninterpreted function \( \text{uniq}_f \) and bind the result to the argument. The congruence property of uninterpreted functions gives us a guarantee that equals cells do not disagree on the results. To make this constraint sound, we have to instantiate equalities for the combinations of cells \( \mathcal{c}_1, \ldots, \mathcal{c}_m \) and \( \mathcal{c}_r^1, \ldots, \mathcal{c}_r^m \).

\[
\text{uniq}_f(\mathcal{c}_i) = \mathcal{c}_r^i \text{ for } 1 \leq i \leq n \quad \text{(Uniqueness)}
\]

3.12 Implementation

We have implemented the symbolic model checker for TLA\(^+\) in Scala. It implements the stages shown in Figure \[3.1\] including the reduction rules introduced in Sections \[3.5–3.10\]. The model checker uses the abstract syntax tree that is built by TLA\(^+\) Tools—the
library that contains the TLA\(^+\) parser SANY and the model checker TLC. Our tool integrates with the SMT solver Z3 by [DB08] via the Java API. We have implemented two techniques: (1) verifying inductive invariants and (2) verifying safety with bounded model checking.

Checking Inductive Invariants In TLA\(^+\), an inductive invariant is a state formula \(\text{Inv}\) that satisfies two conditions: (1) \(\text{Init} \Rightarrow \text{Inv}\), and (2) \(\text{Inv} \land \text{Next} \Rightarrow \text{Inv}'\). Formula \(\text{Inv}'\) is a copy of \(\text{Inv}\), where every variable \(x\) is replaced with its primed version \(x'\). The invariant formula \(\text{Inv}\) usually contains a constraint on the possible values of the variables such as \(x \in 1..10\).

Recall that the formula \(\text{Next}\) is decomposed into a non-deterministic disjunction of symbolic transitions \(T_1 \oplus \ldots \oplus T_m\) in the preprocessing phase (see Section 3.2). Our model checker tests Condition (2) for each transition \(T_i\), that is, it applies the reduction system to the initial state \(\langle \text{Inv} \land T_i \land \neg \text{Inv}' \mid A_0 \mid \nu_0 \mid \Phi_0 \rangle\) and obtains the final state \(\langle c_{\text{final}} \mid A_k \mid \nu_k \mid \Phi_k \rangle\). The tool asks the solver, whether \(\Phi_k \land c_{\text{final}}\) is satisfiable. If this is the case, the tool reports a counterexample to induction, which is obtained from the SMT model. If this is not the case for all \(1 \leq i \leq m\), the inductive invariant holds true.

Finding inductive invariants for TLA\(^+\) specifications is hard. Usually, protocol specifications come with safety properties, which are much simpler to write than inductive invariants. Hence, we have implemented a technique for bounded model checking of such safety properties.

Bounded Model Checking Given a safety property \(P\) and a number \(k \geq 0\), this technique verifies, whether there is a computation of length up to \(k\) that violates the property \(P\) in one of the computation states. Equations (3.4)–(3.5) show a series of reductions that are used to encode a computation of length \(k\). The values of the variables \(\vec{x}'\) computed at step \(i\) are used as the values of the variables \(\vec{x}\) at step \(i+1\). This is done by changing the variable substitution \(\nu_i\) to \(\nu_i[\vec{x} \mapsto \vec{x}', \vec{x}' \mapsto \bot]\).

\[
\begin{align*}
\langle \text{Init}' \mid A_0 \mid \nu_0 \mid \Phi_0 \rangle & \leadsto \ldots \leadsto \langle c_1 \mid A_1 \mid \nu_1 \mid \Phi_1 \rangle \quad (3.4) \\
\langle \text{Next} \mid A_i \mid \nu_i[\vec{x} \mapsto \vec{x}', \vec{x}' \mapsto \bot] \mid \Phi_i \rangle & \leadsto \ldots \leadsto \langle c_{i+1} \mid A_{i+1} \mid \nu_{i+1} \mid \Phi_{i+1} \rangle \text{ for } 1 \leq i \leq k \quad (3.5)
\end{align*}
\]

To check, whether the property \(P\) can be violated after the transition \(i - 1\), the tool rewrites \(\neg P\) as in Equation (3.6). Then, the SMT formula \(\Phi_i \land c_i \land \bigwedge_{1 \leq j \leq i} c_j\) states that the property \(P\) is violated after the transition \(i - 1\). Satisfiability of this formula gives us a counterexample.

\[
\langle \neg P \mid A_i \mid \nu_i \mid \Phi_i \rangle \leadsto \ldots \leadsto \langle c_i \neg P \mid A_i \neg P \mid \nu_i \neg P \mid \Phi_i \neg P \rangle \text{ for } 1 \leq i \leq k \quad (3.6)
\]
3.13 Experiments

In the following, we introduce our experiments with APALACHE and TLC in the TLA+ Toolbox version 1.5.7 [Mic]. Our experiments were run in Grid5000 — a testbed for distributed computing. The experiments were run in parallel using one cluster node of the cluster grvingt (2 CPUs Intel Xeon Gold 6130, 16 cores/CPU, 192GB); each experiment was assigned one core. For simplicity of the setup, we measured wall times. Since many benchmarks run for minutes or hours, we do not consider this imprecision in time measurement to be an issue.

3.13.1 Benchmarks

For most of our examples, we used the benchmarks from the repository TLA+ Examples[1]. The traffic example is given by [Way18]. Table 3.2 shows the benchmarks that we use in the experiments. They range from logical puzzles to concurrent algorithms and fault-tolerant distributed algorithms. The table also lists the values of the parameters, called constants in TLA+, which are used in the experiments. For each benchmark, we give the smallest reasonable value and a larger value.

These benchmarks were previously tried with TLC, some of them contain proofs of safety in TLAPS. Importantly, our modifications to the specifications are minimal. They contain type annotations and, in rare cases, equivalent expressions instead of original complex expressions that would not be handled by our tool otherwise. We neither

---

1The repository TLA+ Examples is available at [github.com/tlaplus/Examples].
introduced simplifications or abstractions in the TLA+ code, in order to run the model checker.

Although the repository contains 64 examples, their complexity varies. Some benchmarks are combinatorial puzzles (e.g. N-Queens, tower-of-hanoi) which are tuned to TLC, while our tool is struggling e.g. with sets of sequences, power sets, and cardinalities. We did not include about 10 trivial teaching examples (e.g. DieHard), because they are no challenge for virtually any model checker. There is a number of Paxos-like algorithms. These are rather complex TLA+ specifications of real distributed algorithms. Both TLC and our tool get stuck after 10-15 steps. We only included the famous Paxos and Raft. Some benchmarks contain recursive operators and rarely-used modules, e.g. Bags. Finally, several benchmarks are only available in the pdf format; we did not try them.

### 3.13.2 Experiments with Inductive Invariants

As explained in Section 3.12, APALACHE checks inductive invariants by reduction to SMT. TLC can also check inductive invariants by state enumeration. We have run both model checkers on a few benchmarks that contained inductive invariants. For each invariant, we have also introduced an invalid invariant candidate: By removing constraints, by introducing arithmetic errors, or by changing constants. This was done

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>time (s)</th>
<th>memory (M)</th>
<th>#tr</th>
<th>#cells</th>
<th>#clauses</th>
<th>time (s)</th>
<th>memory (M)</th>
<th>#states</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bakery-5</td>
<td>1m33s</td>
<td>1.10G</td>
<td>16</td>
<td>25K</td>
<td>131K</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>EWD840-10</td>
<td>5s</td>
<td>687M</td>
<td>4</td>
<td>5.2K</td>
<td>36K</td>
<td>2s</td>
<td>171M</td>
<td>2.0K</td>
</tr>
<tr>
<td>3</td>
<td>bcastByz-4</td>
<td>3s</td>
<td>407M</td>
<td>5</td>
<td>1.7K</td>
<td>10K</td>
<td>2s</td>
<td>401M</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>TwoPhase-7</td>
<td>4s</td>
<td>608M</td>
<td>7</td>
<td>4.8K</td>
<td>23K</td>
<td>2h44m</td>
<td>2.28G</td>
<td>1.14M</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>time (s)</th>
<th>memory (M)</th>
<th>#tr</th>
<th>#cells</th>
<th>#clauses</th>
<th>time (s)</th>
<th>memory (M)</th>
<th>#states</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bakery-5</td>
<td>51s</td>
<td>873M</td>
<td>16</td>
<td>15K</td>
<td>85K</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>EWD840-9</td>
<td>5s</td>
<td>453M</td>
<td>4</td>
<td>2.4K</td>
<td>19K</td>
<td>39s</td>
<td>3.35G</td>
<td>4.47M</td>
</tr>
<tr>
<td>3</td>
<td>EWD840-11</td>
<td>5s</td>
<td>482M</td>
<td>4</td>
<td>2.4K</td>
<td>19K</td>
<td>11m32s</td>
<td>4.41G</td>
<td>92M</td>
</tr>
<tr>
<td>4</td>
<td>EWD840-13</td>
<td>5s</td>
<td>449M</td>
<td>4</td>
<td>2.4K</td>
<td>19K</td>
<td>16h52m</td>
<td>5.55G</td>
<td>1.17B</td>
</tr>
<tr>
<td>5</td>
<td>bcastByz-4</td>
<td>4s</td>
<td>271M</td>
<td>5</td>
<td>463</td>
<td>1.1K</td>
<td>1s</td>
<td>134M</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>bcastByz-10</td>
<td>3s</td>
<td>298M</td>
<td>5</td>
<td>1.2K</td>
<td>5.4K</td>
<td>18m21s</td>
<td>3.40G</td>
<td>16M</td>
</tr>
<tr>
<td>7</td>
<td>TwoPhase-7</td>
<td>6s</td>
<td>483M</td>
<td>7</td>
<td>3.3K</td>
<td>16K</td>
<td>2h47m</td>
<td>2.28G</td>
<td>2.28M</td>
</tr>
<tr>
<td>8</td>
<td>TwoPhase-9</td>
<td>6s</td>
<td>642M</td>
<td>7</td>
<td>4.6K</td>
<td>28K</td>
<td>TO</td>
<td>2.28G</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>TwoPhase-11</td>
<td>7s</td>
<td>737M</td>
<td>7</td>
<td>6.0K</td>
<td>43K</td>
<td>TO</td>
<td>2.27G</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 3.5: The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are satisfied.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>APALACHE</th>
<th>TLC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>memory</td>
</tr>
<tr>
<td>1</td>
<td>Traffic</td>
<td>6s</td>
<td>221M</td>
</tr>
<tr>
<td>2</td>
<td>Prisoners-4</td>
<td>3m19s</td>
<td>355M</td>
</tr>
<tr>
<td>3</td>
<td>Bakery-5</td>
<td>18ms</td>
<td>774M</td>
</tr>
<tr>
<td>4</td>
<td>EWD840-4</td>
<td>56s</td>
<td>1.13G</td>
</tr>
<tr>
<td>5</td>
<td>EWD840-10</td>
<td>13m</td>
<td>1.17G</td>
</tr>
<tr>
<td>6</td>
<td>SimpAlloc-2-2</td>
<td>3s</td>
<td>371M</td>
</tr>
<tr>
<td>7</td>
<td>SimpAlloc-5-3</td>
<td>2h56m</td>
<td>722M</td>
</tr>
<tr>
<td>8</td>
<td>bcastFolk-4</td>
<td>20s</td>
<td>712M</td>
</tr>
<tr>
<td>9</td>
<td>bcastFolk-20</td>
<td>1m9s</td>
<td>1.11G</td>
</tr>
<tr>
<td>10</td>
<td>bcastByz-4</td>
<td>9m14s</td>
<td>1.13G</td>
</tr>
<tr>
<td>11</td>
<td>bcastByz-6</td>
<td>3h00m</td>
<td>1.18G</td>
</tr>
<tr>
<td>12</td>
<td>TwoPhase-3</td>
<td>1m13s</td>
<td>475M</td>
</tr>
<tr>
<td>13</td>
<td>TwoPhase-7</td>
<td>44m</td>
<td>516M</td>
</tr>
<tr>
<td>14</td>
<td>Paxos-3</td>
<td>1h37m</td>
<td>825M</td>
</tr>
<tr>
<td>15</td>
<td>Paxos-5</td>
<td>7h09m</td>
<td>1015M</td>
</tr>
<tr>
<td>16</td>
<td>Raft-5</td>
<td>2h47m</td>
<td>1.18G</td>
</tr>
</tbody>
</table>

As one sees from the few examples, our model checker is fast at proving inductive invariants, while the performance of TLC degrades with larger state spaces. Our model checker is also fast at detecting invariant violation, in the examples with invalid invariant candidates.

It was easy to check the benchmark “bcastByz” for TLC, as the inductive invariant was written for the case when no broadcast occurs in the algorithm, so the number of reachable states is just eight. Notably, TLC cannot check “Bakery” in principle, as it requires one to reason about unbounded integers. Although APALACHE does not support infinite sets, it supports integer constants, so we added a few additional rewriting rules to handle the benchmarks like “Bakery.”
3. TLA⁺ Model Checking Made Symbolic

Table 3.6: The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are violated.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>APALACHE</th>
<th>TLC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>memory</td>
</tr>
<tr>
<td>1</td>
<td>SimpAlloc-5-3</td>
<td>5s</td>
<td>323M</td>
</tr>
<tr>
<td>2</td>
<td>SimpAlloc-3-5</td>
<td>3s</td>
<td>315M</td>
</tr>
<tr>
<td>3</td>
<td>bcastByz-4</td>
<td>2s</td>
<td>254M</td>
</tr>
<tr>
<td>4</td>
<td>bcastByz-12</td>
<td>49s</td>
<td>949M</td>
</tr>
<tr>
<td>5</td>
<td>bcastFolklore-20</td>
<td>2s</td>
<td>301M</td>
</tr>
<tr>
<td>6</td>
<td>Paxos-3</td>
<td>4s</td>
<td>437M</td>
</tr>
<tr>
<td>7</td>
<td>Prisoners-8</td>
<td>4s</td>
<td>416M</td>
</tr>
<tr>
<td>8</td>
<td>Prisoners-10</td>
<td>7s</td>
<td>525M</td>
</tr>
<tr>
<td>9</td>
<td>TwoPhase-5</td>
<td>6s</td>
<td>402M</td>
</tr>
<tr>
<td>10</td>
<td>EWD840-10</td>
<td>18s</td>
<td>753M</td>
</tr>
<tr>
<td>11</td>
<td>EWD840-12</td>
<td>30s</td>
<td>824M</td>
</tr>
</tbody>
</table>

3.13.3 Experiments with Bounded Model Checking

Table 3.3 summarizes the results of the experiments with bounded model checking of safety properties. Table 3.6 summarizes the results of the experiments with the modified specifications that contain buggy behavior. The column “depth” shows the maximum execution length used by our tool as well as the maximum depth reached by TLC while running breadth-first search. The meaning of the other columns is the same as in Table 3.3, see Section 3.13.2. For the small benchmarks we used the diameter bound that was reported by TLC, which does exhaustive state exploration. For the complex benchmarks we used a large enough bound on the length that allowed each experiment to finish within 24 hours. When the depth of APALACHE is smaller than the depth reported by TLC, APALACHE explores a smaller portion of the state space than TLC. For the Raft benchmark, we only report on the experiments with our tool, as TLC has produced an enormous file to store the state exploration queue and exceeded the disk quota of 100 GB in the cluster environment.

In these experiments we check safety properties, e.g., mutual exclusion in case of Bakery and consistency in case of two-phase commit. Specifications of these properties are much smaller than the inductive invariants that would be required for a complete proof with TLAPS.

TLC quickly finishes on the benchmarks with small state spaces, while our tool produces a large set of SMT constraints, independently of the actual number of reachable states. When we supply larger parameter values, the slowdown of our tool is less dramatic than that of TLC. However, as expected, our tool slows down when unrolling longer computations. Usually, it quickly unrolls the computations of length up to 10-15, and then the SMT solver Z3 dramatically slows down when proving unsatisfiability of invariant violation. This is especially noticeable on the specifications of fault-tolerant distributed algorithms such as Paxos and Raft. In these algorithms, after several steps all but few
symbolic transitions become enabled. As a result, proving safety is much harder for Z3, as it has to show unsatisfiability of a formula for all possible schedules of the symbolic transitions. In almost deterministic distributed algorithms such as EWD840, one or two transitions are enabled at the same time, and thus the solver propagates constraints much faster. If we change the safety property to TRUE, that is, APALACHE has to find only whether a symbolic transition is enabled at ith step, Z3 answers the queries in seconds or minutes. We will investigate why such non-determinism and safety properties pose hard problems for Z3 in the future.

3.13.4 Discussion on Performance

Our experiments show a clear advantage of APALACHE over TLC when checking inductive invariants, both in the satisfiable and unsatisfiable case. However, the advantages of our model checker are less pronounced when analyzing safety by bounded model checking. Over 20 years TLC has collected clever heuristics for TLA+. We hope that with the growing number of users, specifications will get tuned to our model checker, as it is now happening with TLC. So far we have found two sources of slowdown in APALACHE:

1. Our benchmarks have non-deterministic control that is hard for SAT/SMT, and
2. The SMT encoding needs solver-specific tuning.

Concerning (1), we considered common patterns in TLA+ specifications. The following code presents a simple benchmark that has non-determinism that is common for TLA+ specifications:

\[
\begin{align*}
\text{Init} & \triangleq x = 0 \\
\text{Next} & \triangleq x' = 1 - x \lor x' = x
\end{align*}
\]

Bounded executions of length \( k \) of this specification pose a challenge for SMT solvers, as they often enumerate \( 2^k \) possible paths without learning. We plan to combine the presented framework with Lipton’s reduction which efficiently eliminates control non-determinism, similar to the work by [KLVW17a].

Concerning (2), there is room for improvement. Unfortunately, SMT solvers are quite sensitive to their input. We believe that the presented framework is solid, though it requires careful tuning of reduction rules for specific SMT solvers. Ideally, we would use a portfolio of SMT solvers and SMT encodings – quantified as well as quantifier-free.

3.14 Summary

We have presented the finite-state symbolic model checker for TLA+ that, similar to the explicit model checker TLC, accepts a range of specifications, which stem from various
application domains. As expected, this permissiveness makes our tool much less efficient in contrast to the model checkers whose input languages and techniques are tailored to specific computational models. Hence, we expect our model checker to be used as the first tool that allows the user to debug their algorithm design before switching to specialized and more efficient tools, or developing a proof with an interactive theorem prover. The example of TLC shows that this happens often in practice. However, TLC does not scale beyond very small parameter values. Hence, we need a symbolic approach to deal with larger parameter spaces.

Since the paper at OOPSLA’19 was published, APALACHE was applied to other case studies:

1. Tendermint blockchain synchronization \cite{BBK+20b},
2. A Tendermint light client \cite{BBK+20a}, and
3. Model checking of Tendermint safety \cite{Sys}.
CHAPTER 4

Soundness of the Reduction from TLA$^+$ to SMT

In this chapter, we prove termination and soundness of the reduction from TLA$^+$ to SMT presented in Chapter 3. We start this chapter with definitions of models and extended models in KerA$^+$. Next, we outline how to prove soundness of our systems. Then, we provide detailed proofs.

In addition, we present how to define many TLA$^+$ operators in KerA$^+$ and provide additional reduction rules from TLA$^+$ to SMT.

This chapter presents the additional definitions and the extended proofs of the results in the paper at OOPSLA’19 [KKT19].

4.1 Models and extended models in KerA$^+$

In this section, we first describe the structure of models in KerA$^+$. Then, we formulate interesting properties about finite structures in KerA$^+$ models. Finally, we present the definition of extended structures.

4.1.1 Models in KerA$^+$

We first introduce sets of objects, and then interpret the meaning of functions and predicates in KerA$^+$. The definition of model $\mathcal{M} = \langle D, I \rangle$ in KerA$^+$ is similar to ones in many-sorted first-order logic, e.g., as given by [Mer12]. Intuitively, a model is a pair $\mathcal{M} = \langle D, I \rangle$ such that

1. $D$ is a universal domain. It is a union of disjoint sets $D_1, \ldots, D_n$, each $D_i$ contains objects of type $\tau_i$. 
2. $I$ is an interpretation. It assigns values from the domain to constants and functions in $\text{KERA}^+$.

For more details, a model $\mathcal{M} = \langle \mathcal{D}, I \rangle$ has the following kinds of domains:

- Every sort $\tau$ has a non-empty set $\mathcal{D}_\tau$. The universal domain $\mathcal{D}$ is a union of disjoint sets $\mathcal{D}_{\tau_1}, \ldots, \mathcal{D}_{\tau_k}$ where $\tau_1, \ldots, \tau_k$ are sorts.
- $\mathcal{D}_{\text{Int}}, \mathcal{D}_{\text{Bool}},$ and $\mathcal{D}_{\text{Name}}$ are respectively sets of integers, Boolean constants, and unique names. A name is a unique string that is written using the TLA$^+$ syntax “abc”.
- Set domain $\mathcal{D}_{\text{Set}[\tau]}$ contains a special object $\emptyset$ for the empty set and objects in form of $\{o_1, \ldots, o_n\}$ that is an enumerative set, where $o_1, \ldots, o_n$ are objects in $\mathcal{D}_\tau$.
- Function domain $\mathcal{D}_{\tau_1 \rightarrow \tau_2}$ contains sets of ordered pairs where every set $s_{\text{fun}}$ represents a function object. Moreover, we require two following constraints on $\mathcal{D}_{\tau_1 \rightarrow \tau_2}$.
  - For every function object $s_{\text{fun}}$, for every pair $(a, res)$ in $s_{\text{fun}}$, objects $a$ and $res$ must be in domains $\mathcal{D}_{\tau_1}$ and $\mathcal{D}_{\tau_2}$, respectively.
  - For every function object $s_{\text{fun}}$, for every pair of its members $(a_1, res_1)$ and $(a_2, res_2)$, if $a_1 = a_2$, then $res_1 = res_2$.
- Tuple domain $\mathcal{D}_{\tau_1 \times \cdots \times \tau_n}$ contains (mathematical) tuple objects, i.e., $\langle o_1, \ldots, o_n \rangle$ where $o_i$ is an object in domain $\mathcal{D}_i$ for every $1 \leq i \leq n$.
- Record domain $\mathcal{D}_{\langle nm_1: \tau_1, \ldots, nm_n: \tau_n \rangle}$ contains record objects that are in form of $[nm_1 \mapsto o_1, \ldots, nm_n \mapsto o_n]$ where $nm_i$ is a name and $o_i$ is an object in domain $\mathcal{D}_i$ for every $1 \leq i \leq n$.

The interpretation $I$ interprets operators in $\text{KERA}^+$ which have counterparts in TLA$^+$ in the standard way [Mer08].

- Every variable $x$ with type $\tau$ is interpreted as an object in $\mathcal{D}_\tau$, i.e., $\llbracket x \rrbracket^\mathcal{M} = I(x)$.
- Every uninterpreted function $f$ with type $\tau$ is interpreted as an object in $\mathcal{D}_\tau$, i.e., $\llbracket f \rrbracket^\mathcal{M} = I(f)$.

Now we define the meaning of the operators highlighted in Table 3.1 of Chapter 3 and operator domain in $\text{KERA}^+$. As usual, we use the notation $\llbracket e \rrbracket^\mathcal{M}$ to denote the value of a $\text{KERA}^+$ expression in model $\mathcal{M}$.

- Expression $x' \in S$ is interpreted as a constraint that $\llbracket x' \rrbracket^\mathcal{M}$ is a member of set $\llbracket S \rrbracket^\mathcal{M}$. Formally, we have $\llbracket x' \rrbracket^\mathcal{M} \in \llbracket S \rrbracket^\mathcal{M}$. 

68
4.1. Models and extended models in $\text{KerA}^+$

- Expression $\text{FROM} c_1, \ldots , c_n$ by $\theta$ is treated as an uninterpreted function with a constraint on its values. $\llbracket \text{FROM} c_1, \ldots , c_n \text{ by } \theta \rrbracket^M$ is interpreted as an object in $D_\tau$ where $\tau$ is a type of $c_i$ for every $1 \leq i \leq n$ such that the following constraint holds

$$\bigwedge_{i=1}^{n} \llbracket \theta \rrbracket^M = \llbracket i \rrbracket^M \rightarrow \llbracket \text{FROM} c_1, \ldots , c_n \text{ by } \theta \rrbracket^M = \llbracket c_i \rrbracket^M$$

Notice that while we have to rewriting rules $\text{FROMBasic}$ and $\text{FROMSet}$, the meaning of the operator $\text{FROM} c_1, \ldots , c_n$ by $\theta$ does not depend on the type of $c_i$.

- Expression $e_1 \oplus \ldots \oplus e_n$ is treated as a standard logical disjunction. Non-deterministic disjunctions are used to optimize symbolic transitions but their meaning from the model viewpoint is the same as standard logical disjunctions.

  - If $I$ maps a function $\text{fun}$ to a set $\{\langle a_1, \text{res}_1 \rangle, \langle a_2, \text{res}_2 \rangle, \ldots \}$, then $\llbracket \text{DOMAIN fun} \rrbracket^M$ is mapped to a set $\{a_1, a_2, \ldots \}$.

  - If $I$ maps a tuple $\text{tup}$ to an object $\langle o_1, \ldots , o_n \rangle$, then $\llbracket \text{DOMAIN tup} \rrbracket^M$ is mapped to a set $\{1, \ldots , n\}$.

  - If $I$ maps a tuple $\text{rcd}$ to an object $[num \mapsto o_1, \ldots , \mapsto o_n]$, then $\llbracket \text{DOMAIN rcd} \rrbracket^M$ is mapped to a set $\{nm_1, \ldots , nm_n\}$.

The satisfiability of a Boolean expression in $\text{KerA}^+$ is defined in the standard way [Mer08].

4.1.2 Finite structures

In our work, the specification parameters are fixed. Thus, every $\text{KerA}^+$ expression “intuitively” defines only finite values. We formalize this intuition by introducing finite structures and showing that every $\text{KerA}^+$ expression $e$ defines a finite structure, as soon as the constants in $e$ are interpreted as finite structures (see Proposition 4.1.1).

For a model $M = \langle D, I \rangle$, a value $v \in D$ is called a finite structure, if one of the following holds:

- Value $v$ has type Int, Bool, or Name,
- Value $v$ is a finite set, whose elements are finite structures,
- Value $v$ is a function $f : S \rightarrow T$ such that $S$ and $T$ are finite structures, or
- Value $v$ is a record, a tuple, or a finite sequence, and $v$’s elements are finite structures.

Proposition 4.1.1 Let $e$ be a $\text{KerA}^+$ expression, and $M = \langle D, I \rangle$ be a model. If $I$ interprets all constants and free variables in $e$ as finite structures, then the interpretation of $e$ is a finite structure.
As expected, we call a model $M = \langle D, I \rangle$ finite, if every value $v \in D$ is a finite structure. Finally, given a state $\langle e \mid A \mid \nu \mid \Phi \rangle$ of the reduction system, a model $M = \langle D, I \rangle$ is suitable for the state, if the expression $e$ and the constraint $\Phi$ can be interpreted with $M$.

### 4.1.3 Extended Models in Ker$A^+$

Assume that $s_{before} \xrightarrow{r} s_{after}$ is a reachable transition in our reduction system, and we already have a model $M_{before}$ for $s_{before}$. The applied reduction rule may introduce fresh cells and expressions which do not appear in $s_{before}$. Therefore, a suitable model $M_{after}$ for $s_{after}$ should assign values to fresh ones. There are many ways to define $M_{after}$. In the following, we describe a particular way to define $M_{after}$ by extending $M_{before}$ with mappings for fresh cells and expressions in $s_{before}$. The models extended in this way are used to prove the soundness of our reduction system.

**Definition 4.1.1** A $\text{Ker}A^+$ model $M_{after} = \langle D_{after}, I_{after} \rangle$ suitable for state $s_{after}$ is called an extension of the model $M_{before} = \langle D_{before}, I_{before} \rangle$ suitable for state $s_{before}$ if the follow hold:

- Some rewriting rule $r$ is applicable to $e_{before}$. Moreover, its application to $e_{before}$ replaces a sub-expression $e_{sub}$ in $e_{before}$ by a fresh cell $c_{\text{fresh}}$, and generates a new expression $e_{after}$.
- Both $M_{after}$ and $M_{before}$ have the same domain, i.e. $D_{after} = D_{before}$.
- If rule $r$ is a big step which uses small rewriting steps inside, then $I$ is inductively constructed through such small steps, and based on $r$.

- Case (FROMBASIC): The interpretation $I_{after}$ is extended from $I_{before}$ with a new mapping $[c_{\text{res}}]_{M_{after}} = [\text{FROM}c_1, \ldots, c_n \text{ BY } \theta]_{M_{after}}$.
- Case (FROMSET): First, for every $1 \leq i \leq n$, for every $m_i \leq j \leq m$, we have $[c_i^j]_{M_{after}} = [c_i^{m_i}]_{M_{after}}$. Then, we have $[c_{\text{res}}]_{M_{after}} = [\text{FROM}c_1, \ldots, c_n \text{ BY } \theta]_{M_{after}}$ and $[c_i^j]_{M_{after}} = [\text{FROM}c_i^1, \ldots, c_i^n \text{ BY } \theta]_{M_{after}}$.
- Cases (FUNEXC) and (FUNDOM): The interpretation $I_{after}$ is the same as $I_{before}$.
- Case (ITE): If $[p]_{M_{before}} = \text{TRUE}$, then The interpretation $I_{after}$ is similar to $I_{before}$, but has additional mappings $[\theta]_{M_{after}} \rightarrow 1$, $[c_{\text{res}}]_{M_{after}} \rightarrow [c_1]_{M_{after}}$. Otherwise, these mappings $[\theta]_{M_{after}} \rightarrow 2$, $[c_{\text{res}}]_{M_{after}} \rightarrow [c_2]_{M_{after}}$ are used.
- Case (FCNCTOR): Assume that $e_{before} \triangleq [x \in S \mapsto e]$, and $e_{after} = c_f$, and $c_f \xrightarrow{\theta} c_{\text{rel}}$. Then, $I_{after}$ is similar to $I_{before}$, but has new mappings $[c_f]_{M_{after}} \rightarrow [[x \in S \mapsto e]]_{M_{after}}$, and $[\text{funrel}(c_f)]_{M_{after}} = [c_{\text{rel}}]_{M_{after}}$.
- Case (FCNAPP): Assume $e_{before} = c_f[c_{\text{arg}}]$, and $e_{after} = c_f$, and $c_f \xrightarrow{\theta} c_{\text{rel}}$. The interpretation $I_{after}$ is extended from $I_{before}$ with new mappings for $\theta$.
and \(c_1, \ldots, c_n\) by \(\theta\). If \(\llbracket c_{\text{arg}} \rrbracket^m_{\text{after}} \notin \llbracket f \rrbracket^m_{\text{before}}\), then \(\llbracket \theta \rrbracket^m_{\text{after}} = 0\). Otherwise, \(\llbracket \theta \rrbracket^m_{\text{after}}\) is mapped to some integer \(i\) such that \(1 \leq i \leq n\), and \(\llbracket c_i \rrbracket^m_{\text{before}} = \llbracket \langle \text{arg}, c_f | \text{arg} \rangle \rrbracket^m_{\text{before}}\).

- **Case (setcard):** The interpretation \(I_{\text{after}}\) is extended with additional mappings for \(k_i\) such that formulas (1) and (2) are still correct.

- **Case (asgn):** If \(\llbracket c_s \rrbracket^m_{\text{after}} = \emptyset\), we extend \(M_{\text{after}}\) from \(M_{\text{before}}\) such that \(\llbracket \theta \rrbracket^m_{\text{after}} = 0\), and \(\llbracket c \rrbracket^m_{\text{after}} = \llbracket x' \rrbracket^m_{\text{after}}\), and both \(\llbracket c \rrbracket^m_{\text{after}}, \llbracket x' \rrbracket^m_{\text{after}}\) are arbitrary. Otherwise, we have that \(1 \leq \llbracket \theta \rrbracket^m_{\text{after}} \leq n\), and both \(\llbracket c \rrbracket^m_{\text{after}} = \llbracket x \rrbracket^m_{\text{after}}\), and both \(\llbracket c \rrbracket^m_{\text{after}}, \llbracket x \rrbracket^m_{\text{after}}\) are arbitrary.

- **Case (choose):** We extend \(M_{\text{after}}\) from \(M_{\text{before}}\) such that \(0 \leq \llbracket \theta \rrbracket^m_{\text{after}} \leq n\), and \(\llbracket c_{\text{F}} \rrbracket^m_{\text{after}} = \llbracket \{ x \in S : p \} \rrbracket^m_{\text{after}}\), and \(\llbracket \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \rrbracket^m_{\text{after}} = \llbracket c_{\text{res}} \rrbracket^m_{\text{after}}\), and \(\llbracket \text{choose}_e (c_{\text{F}}) \rrbracket^m_{\text{after}} = \llbracket c_{\text{res}} \rrbracket^m_{\text{after}}\).

- **Other cases:** the interpretation \(I_{\text{after}}\) is similar to \(I_{\text{before}}\), but has an additional mapping \(\llbracket c_{\text{fresh}} \rrbracket^m_{\text{after}} \rightarrow \llbracket e_{\text{sub}} \rrbracket^m_{\text{before}}\).

### 4.2 Soundness and Termination: Overview

In this section, we discuss two important properties of the reduction system: termination and soundness. These results are formalized in Theorems 4.2.1 and 4.2.2. We also introduce the invariants that are used to show soundness of the reduction. The final result guarantees that the constraints produced by the reduction system belong to the SMT theories.

**Theorem 4.2.1** Every sequence of ARS reductions \(s_0 \rightarrow s_1 \rightarrow \ldots\) is finite. In other words, the reduction process terminates.

To prove Theorem 4.2.1 we define a partial order on KERA\(^+\) expressions and show that every reduction rule produces smaller expressions.

**Theorem 4.2.2** Let \(s_0 \rightarrow \ldots \rightarrow s_m\) be a sequence of states produced by an abstract reduction system, and \(s_i = \langle e_i | A_i | \nu_i | \Phi_i \rangle\) for \(1 \leq i \leq m\). Assume that \(e_0\) is a formula, that is, it has type \(\text{Bool}\). The formula \(e_0\) is satisfiable if and only if the constraint \(e_m \land \Phi_m\) is satisfiable.

Note that if the reduction system terminates without an error, then the terminal expression \(e_m\) in Theorem 4.2.2 is a constant. Moreover, the reductions produce constraints that are compatible with SMT solvers [BFT17].

\(^1\)We prove that there exists such integer in Lemma 4.3.17
Proposition 4.2.1 Let $s_0 \leadsto \ldots \leadsto s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = \langle e_i \mid A_i \mid \nu_i \mid \Phi_i \rangle$ for $1 \leq i \leq m$. Then, every formula $\Phi_i$ is a quantifier-free first-order logic formula over uninterpreted functions and integer arithmetic.

In the following, we give the idea of our proof of Theorem 4.2.2. Detailed proofs are presented in the next section. We prove the theorem by showing that the abstract reduction system satisfies seven invariants on the reachable states and transitions of the system. As usual, a state $s_m$ of the reduction system is reachable, if there is a finite sequence of rewriting transitions $s_0 \leadsto \ldots \leadsto s_m$ from an initial state $s_0$ leading to $s_m$. Similarly, a transition is reachable, if it originates from a reachable state.

We observe that every reduction rule transforms a KERA$^+$ expression $e_{\text{before}}$ in an expression $e_{\text{after}}$ in a special way. In particular, a model $M_{\text{after}}$ of $e_{\text{after}}$ differs from a model $M_{\text{before}}$ of $e_{\text{before}}$ in that $M_{\text{after}}$ has additional constants. Hence, we call $M_{\text{after}}$ an extended model of $M_{\text{before}}$.

Invariants of the Reduction System. In order to prove soundness of the translation to SMT, we formulate six invariants on the reachable states and transitions of the abstract reduction system. Proposition 4.2.2 ensures that all invariants 4.2.1-4.2.7 are preserved by every sequence of transitions.

Invariant 4.2.1 states that our reduction system produces only well-typed expressions:

**Invariant 4.2.1** In every reachable state $\langle e \mid A \mid \nu \mid \Phi \rangle$ of the ARS, the expression $e$ is well-typed.

Invariant 4.2.2 gives us a relation between the arenas and the Boolean constants that are introduced for the arena edges in the constraint $\Phi$:

**Invariant 4.2.2** In every reachable state $\langle e \mid A \mid \nu \mid \Phi \rangle$ of the ARS, the following holds:

1. Every cell $c$ appears in either the expression $e$ or the formula $\Phi$ if and only if it appears in $A$.

2. Arena $A$ has an edge $(c_{\text{set}}, i, c_{\text{elem}})$ if and only if the formula $\Phi$ contains the constant $en(c_{\text{set}}, i, c_{\text{elem}})$.

Invariant 4.2.3 ensures that the reduction rules produce suitable models, and Invariant 4.2.4 refers to the reverse direction:

**Invariant 4.2.3** Let $s_{\text{before}} \leadsto s_{\text{after}}$ be a reachable transition in the ARS, and $M_{\text{before}}$ be a suitable model for $s_{\text{before}}$. An extended structure $M_{\text{after}}$ from $M_{\text{before}}$ is also suitable for $s_{\text{after}}$. 
4.3. Detailed proofs

Invariant 4.2.4 Let \( s_{\text{before}} \rightarrow s_{\text{after}} \) be a reachable transition in the ARS, and \( \mathcal{M}_{\text{after}} \) be a suitable model for \( s_{\text{after}} \). Then, \( \mathcal{M}_{\text{after}} \) is also suitable for \( s_{\text{before}} \).

Invariant 4.2.5 states that the arena is preserving an overapproximation of every set cell:

**Invariant 4.2.5** Let \( \langle e \mid A \mid v \mid \Phi \rangle \) be a reachable state of the ARS, and \( \mathcal{M} \) be its extended model. Assume that \( c_{\text{set}} \) is a set cell in the arena \( A \). Then, the following holds:

1. Assume that \( c_{\text{set}} \rightarrow_A c_1, \ldots, c_n \), for some \( n \geq 0 \), and \( c_{\text{set}} \) is introduced by a rule different from (FROMSET). Then, the following holds: \( \lbrack c_{\text{set}} \rbrack^\mathcal{M} \subseteq \{ \lbrack c_1 \rbrack^\mathcal{M}, \ldots, \lbrack c_n \rbrack^\mathcal{M} \} \).

2. Assume that \( c_{\text{set}} \) is a reduction of the expression \( \text{FROM} c_1, \ldots, c_n \) by \( \theta \) with \( 1 \leq \theta \leq n \) and \( c_{\text{set}} \rightarrow_A c^1_{\text{pick}}, \ldots, c^n_{\text{pick}} \). Then, the following holds:

\[
\lbrack c_{\text{set}} \rbrack^\mathcal{M} \subseteq \{ \lbrack c^1_{\text{pick}} \rbrack^\mathcal{M}, \ldots, \lbrack c^n_{\text{pick}} \rbrack^\mathcal{M} \}
\]

Invariant 4.2.6 states that a function cell is always pointing to the associated relation cell:

**Invariant 4.2.6** Let \( \langle e \mid A \mid v \mid \Phi \rangle \) be a reachable state of the ARS. Assume that \( c_{\text{f}} \) is a function cell of type \( \tau_1 \rightarrow \tau_2 \) in the arena \( A \). Then, there is a cell \( c_{\text{rel}} \) of type \( \text{Set}[\tau_1 \times \tau_2] \) such that the function cell is pointing to it: \( c_{\text{f}} \rightarrow_A c_{\text{rel}} \).

Finally, Invariant 4.2.7 is about the equality between a function cell \( c_{\text{f}} \) in the arena and its set representation constructed based on the corresponding cell \( c^\prime_{\text{rel}} \).

**Invariant 4.2.7** Let \( \langle e \mid A \mid v \mid \Phi \rangle \) be a reachable state of the ARS, and \( \mathcal{M} \) be its extended model. Assume that \( c_{\text{f}} \) is a function cell, and \( c_{\text{f}} \rightarrow_A c_{\text{rel}} \). Then, it follows that the set \( \lbrack c_{\text{rel}} \rbrack^\mathcal{M}_{\text{after}} \) is equal to the set \( \lbrack \{ (x, f(x)) : x \in \text{DOMAIN } f \} \rbrack^\mathcal{M}_{\text{after}} \).

The following propositions state that the above introduced invariants hold true:

**Proposition 4.2.2** Let \( s_0 \rightarrow \ldots \rightarrow s_m \) be a sequence of states produced by an abstract reduction system. Then, Invariants 4.2.3 and 4.2.4 are preserved by every transition \( s_i \rightarrow s_{i+1} \) for every \( 0 \leq s < m \). Moreover, Invariants 4.2.1, 4.2.2, and 4.2.5–4.2.7 are preserved by every state \( s_j \) for every \( 0 \leq j \leq m \).

4.3 Detailed proofs

4.3.1 Termination

In order to prove Theorem 4.2.1 we define a partial order \( \prec \) on \( \text{KER}^+ \) expressions such that after applying a reduction rule \( r \) to \( e_{\text{before}} \), except the rule (CHOOS), we should have a “smaller” expression \( e_{\text{after}} \), i.e. \( e_{\text{after}} \prec e_{\text{before}} \). Definition 4.3.1 describes the partial order \( \prec \), and the desired property is obtained by Lemma 4.3.1.
4. Soundness of the Reduction from TLA$^+$ to SMT

**Definition 4.3.1** Let $\text{sum}(e)$ be a function to the sum of branch lengths in $e$’s construction tree, and $\text{count}(e)$ be the number of sub-expressions that are not cells in $e$. We introduce a new relation $\prec$ between expressions $e_1, e_2$ such that $e_1 \prec e_2$ if and only if $\text{sum}(e_1) > \text{sum}(e_2)$, or $\text{sum}(e_1) = \text{sum}(e_2) \land \text{count}(e_1) < \text{count}(e_2)$.

**Lemma 4.3.1** Let $e_{\text{before}}$ be an arbitrary KERA$^+$ expression without any operator choose, and $e_{\text{after}}$ be the result of applying an arbitrary rewriting rule $r$ to $e_{\text{before}}$. We have $e_{\text{after}} \prec e_{\text{before}}$.

**Proof.** Table 4.1 shows the relationship between $e_{\text{before}}$ and $e_{\text{after}}$ with functions sum and count. Because $\text{sum}(e) \geq 1$ and $\text{count}(e) \geq 0$ for any expression $e$, we always have $e_{\text{before}} \prec e_{\text{after}}$ when applying any rewriting rules, except choose. However, we assume that there exists no choose in $e_{\text{before}}$, and therefore we have that $e_{\text{after}} \prec e_{\text{before}}$.

**Table 4.1:** The values of sum and count on $e_{\text{before}}$ and $e_{\text{after}}$

<table>
<thead>
<tr>
<th>Rewriting rules</th>
<th>Comparisons</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool, Int</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}})$</td>
</tr>
<tr>
<td></td>
<td>$\text{count}(e_{\text{before}}) = \text{count}(e_{\text{after}}) + 1$</td>
</tr>
<tr>
<td><strong>IntCmp, IntArith</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) + 2$</td>
</tr>
<tr>
<td><strong>RedArg</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) + (\sum_{k=1}^{n} \text{sum}(\text{arg}_k) - 1)$</td>
</tr>
<tr>
<td></td>
<td>$\text{count}(e_{\text{before}}) = \text{count}(e_{\text{after}}) + n$</td>
</tr>
<tr>
<td><strong>Enum</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) - n$</td>
</tr>
<tr>
<td><strong>Filter</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) + 3 + \text{sum}(p)$</td>
</tr>
<tr>
<td><strong>Union</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) + 1$</td>
</tr>
<tr>
<td><strong>Map</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) + 3 + \text{sum}(e)$</td>
</tr>
<tr>
<td><strong>FromBasic, FromSet</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) + 1$</td>
</tr>
<tr>
<td><strong>Choose</strong></td>
<td>$\text{sum}(e_{\text{before}}) = \text{sum}(e_{\text{after}}) - 1$</td>
</tr>
</tbody>
</table>

**Theorem 4.2.1.** Every sequence of ARS reductions $s_0 \to s_1 \to \ldots$ is finite. In other words, the reduction process terminates.

**Proof.** We first prove the lemma in the case where there exists no operator choose in $e_0$. It is easy to check that $e_i$ does not contain any operator choose for every $i$. By Lemma 4.3.1, it follows that $e_{i+1} \prec e_i$. Thus, the rewriting procedure must stop after finite steps. We now turn to the case where there exist operators choose in $e_0$. Because $e_{\text{init}}$ is a finite string, there should be finitely many application of choose. Assume that choose appears $k$ times in $e_{\text{init}}$. Then we know there exists no choose in $e_k$ because the $k$-first rewriting rules are to remove choose. We now apply Lemma 4.3.1 to $s_k$. Thus, the rewriting procedure for $e_k$ must stop after finite steps. Combining the above cases, we obtain the lemma.
4.3. Detailed proofs

4.3.2 SMT encoding

Proposition 4.2.1. Let $s_0 \rightarrow \ldots \rightarrow s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = (e_i \mid A_i \mid \nu_i \mid \Phi_i)$ for $1 \leq i \leq m$. Then, every formula $\Phi_i$ is a quantifier-free first-order logic formula over uninterpreted functions and integer arithmetic.

Proof. We prove the proposition by induction on transition steps. The basis is obviously correct because $\Phi_0$ is empty. It is easy to check that every reduction rule always produces contraints that are quantifier-free first-order logic formulas over uninterpreted functions and integer arithmetic. Therefore, Proposition 4.2.1 is correct.

4.3.3 Miscellaneous

We first present technical Lemmas 4.3.2–4.3.3 in first-order logic which allow us to remove quantifiers. Then, we present Lemma 4.3.4 which is related to a substitution. These lemmas are used in the proof of others in Sections 4.3.4 and 4.3.5.

Lemma 4.3.2 Let $S$ be a $\mathsf{KER}^+\tau$ set expression with a type $\mathsf{Set}[\tau]$, and $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ a $\mathsf{KER}^+\tau$ structure for $S$. Assume that a finite set $\{o_1, \ldots, o_n\}$ is an over-approximation of $[S]^\mathcal{M}$ where $o_i \in \mathcal{D}_\tau$ for every $1 \leq i \leq n$. Then, the following formulas hold true under $\mathcal{M}$:

$$\forall x \in S \cdot p(x) \xrightarrow{\mathcal{M}} \bigwedge_{i=1}^n o_i \in [S]^\mathcal{M} \Rightarrow [p(o_i)]^\mathcal{M} (4.1)$$

$$\exists x \in S \cdot p(x) \xrightarrow{\mathcal{M}} \bigvee_{i=1}^n o_i \in [S]^\mathcal{M} \land [p(o_i)]^\mathcal{M} (4.2)$$

Proof. Assume that in both formulas, $S$ has type $\mathsf{Set}[\tau]$, and thus, its elements have type $\tau$. The proof of the formula (6) falls naturally into two parts.

- Case $\forall x \in S \cdot p(x) \xrightarrow{\mathcal{M}} \bigwedge_{i=1}^n (o_i \in [S]^\mathcal{M} \Rightarrow [p(o_i)]^\mathcal{M})$:

Assume that $\forall x \in S \cdot (p(x) \xrightarrow{\mathcal{M}}$ is true. For every value $v \in \mathcal{D}_\tau$ such that $v \in [S]^\mathcal{M}$, we have $[p(x)[v/x]]^\mathcal{M} = \text{true}$. Then, it follows that $\bigwedge_{i=1}^n (o_i \in [S]^\mathcal{M} \Rightarrow [p(o_i)]^\mathcal{M})$ is true.

- Case $\forall x \in S \cdot (p(x) \xrightarrow{\mathcal{M}} \bigvee_{i=1}^n (o_i \in [S]^\mathcal{M} \Rightarrow [p(o_i)]^\mathcal{M})$:

Assume that $\bigvee_{i=1}^n (o_i \in [S]^\mathcal{M} \Rightarrow [p(o_i)]^\mathcal{M})$ is true. It follows that if $o_i \in [S]^\mathcal{M}$, then $[p(o_i)]^\mathcal{M}$ is true for every $o_i$. Because $[S]^\mathcal{M}$ is a subset of the set $\{o_1, \ldots, o_n\}$, it follows that $[p(x)[v/x]]^\mathcal{M}$ is true for every member $v$ of $[S]^\mathcal{M}$. Hence, we have that $\forall x \in S \cdot p(x) \xrightarrow{\mathcal{M}}$ is correct.

Combining two above cases, we have that the formula (6) is correct. Similar arguments apply to the proof of the formula (7).
4. Soundness of the Reduction from TLA$^+$ to SMT

**Lemma 4.3.3** Let $S$ and $T$ be KERA$^+$ set expressions with a type $\text{Set}[\tau]$. Let $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ a KERA$^+$ structure for $S$, $T$. Assume that a finite set $\{o_1, \ldots, o_n\}$ is an over-approximation of both $[S]^\mathcal{M}$ and $[T]^\mathcal{M}$ where $o_i \in \mathcal{D}$ for every $1 \leq i \leq n$. Then, the following formula holds true under $\mathcal{M}$:

$$[S = T]^\mathcal{M} \iff \bigwedge_{i=1}^n \left( o_i \in [S]^\mathcal{M} \iff o_i \in [T]^\mathcal{M} \right)$$

**Proof.** We have $[S = T]^\mathcal{M} \iff (\forall x \in S . x \in T)^\mathcal{M} \land (\forall x \in T . x \in S)^\mathcal{M}$ by the axiom of extensionality. By the lemma assumption and Lemma 4.3.2 it follows

$$[\forall x \in S . x \in T]^\mathcal{M} \iff \bigwedge_{i=1}^n \left( o_i \in [S]^\mathcal{M} \implies o_i \in [T]^\mathcal{M} \right) , \text{ and }$$

$$[\forall x \in T . x \in S]^\mathcal{M} \iff \bigwedge_{i=1}^n \left( o_i \in [T]^\mathcal{M} \implies o_i \in [S]^\mathcal{M} \right)$$

Combining these results, we have $[S = T]^\mathcal{M} \iff \bigwedge_{i=1}^n \left( o_i \in [S]^\mathcal{M} \iff o_i \in [T]^\mathcal{M} \right)$. ■

**Lemma 4.3.4** Let $\text{exp}, e_{\text{sub}}, e_{\text{new}}$ be well-typed KERA$^+$ expressions such that (i) $e_{\text{sub}}$ is a subexpression of $\text{exp}$, and (ii) every free variable in $e_{\text{sub}}$ is also free in $\text{exp}$, and (iii) $e_{\text{new}}$ is a constant which does not appear in $\text{exp}$. Let $\mathcal{M} = (\mathcal{D}, \mathcal{I})$ be a KERA$^+$ model such that $[e_{\text{new}}]^\mathcal{M} = [e_{\text{sub}}]^\mathcal{M}$. Then, the following equality holds true: $[\text{exp}]^\mathcal{M} = [\text{exp}[e_{\text{new}}/e_{\text{sub}}]]^\mathcal{M}$.

**Proof.** We prove this lemma by induction on the formula structure. In the base case, $\text{exp}$ is either a constant, or a free variable. In this case, $e_{\text{sub}}$ coincides with $\text{exp}$. Then, $\text{exp}[e_{\text{new}}/e_{\text{sub}}]$ is syntactically the same as $e_{\text{new}}$, which gives us the required equality.

Now consider the case when $\text{exp}$ is a $k$-ary term $f(e_1, \ldots, e_k)$ in KERA$^+$. By the induction hypothesis, we know $[e_i]^\mathcal{M} = [e_i[e_{\text{new}}/e_{\text{sub}}]]^\mathcal{M}$. By function congruence, we have that $[[f(e_1, \ldots, e_k)]^\mathcal{M} = [[f[e_1, \ldots, e_k]]^\mathcal{M} = [e_{\text{new}}/e_{\text{sub}}]^\mathcal{M}$. When $\text{exp}$ is a formula, that is, it is constructed using the Boolean connectives, the proof is exactly the same. ■

4.3.4 Invariants

**Invariant 4.2.1** In every reachable state $\langle e \mid A \mid \nu \mid \Phi \rangle$ of the ARS, the expression $e$ is well-typed.

**Lemma 4.3.5** Let $s_0 \rightarrow \ldots \rightarrow s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = \langle e_k \mid A_k \mid \nu_k \mid \Phi_k \rangle$ for $1 \leq k \leq m$. Assume that $e_0$ is well-typed. Then, Invariant 4.2.1 is preserved by every state $s_k$ for every $0 \leq k \leq m$. 76
4.3. Detailed proofs

\textbf{Proof.} We prove the lemma by induction on transition steps. By the lemma assumption, the basis is clearly correct. We now prove the induction step. Assume that a sub-expression $e_{\text{sub}}$ in $e_i$ is replaced with an expression $\text{exp}$. It is easy to check that the expressions $\text{exp}$ and $e_{\text{sub}}$ have the same type. It follows that $e_{i+1}$ is well-typed. Therefore, Lemma 4.3.5 is correct. \hfill \blacksquare

\textbf{Invariant 4.2.2.} In every reachable state $(e \mid A \mid \nu \mid \Phi)$ of the ARS, the following holds:

1. Every cell $c$ appears in either the expression $e$ or the formula $\Phi$ if and only if it appears in $A$.
2. Arena $A$ has an edge $(c_{\text{set}}, i, c_{\text{elem}})$ if and only if the formula $\Phi$ contains the constant $cn(c_{\text{set}}, i, c_{\text{elem}})$.

\textbf{Lemma 4.3.6} Let $s_0 \leadsto \ldots \leadsto s_m$ be a sequence of states produced by an abstract reduction system where $s_0$ is an initial state. Then, Invariant 4.2.2 is preserved by every state $s_k$ for every $0 \leq k \leq m$.

\textbf{Proof.} We prove the lemma by induction on transition steps. The basis is obviously correct since both the arena $A_0$ and the constraint $\Phi_0$ are empty. It is easy to check that the induction step is true by the definition of the reduction rules. Therefore, Lemma 4.3.6 is correct. \hfill \blacksquare

\textbf{Invariant 4.2.3.} Let $s_{\text{before}} \leadsto s_{\text{after}}$ be a reachable transition in the ARS, and $M_{\text{before}}$ be a suitable model for $s_{\text{before}}$. An extended structure $M_{\text{after}}$ from $M_{\text{before}}$ is also suitable for $s_{\text{after}}$.

\textbf{Lemma 4.3.7} Let $s_0 \leadsto \ldots \leadsto s_m$ be a sequence of states produced by an abstract reduction system where $s_0$ is an initial state. Then, Invariant 4.2.3 is preserved by every reachable transition $s_k \leadsto s_{k+1}$ for every $0 \leq k < m$.

\textbf{Proof.} We prove the lemma by induction on transition steps. The basis is obviously true because the lemma assumption. Recall the construction of extended models in Definition 4.1.1 is guided by the reduction rules. Therefore, it is easily seen that the induction step is correct. \hfill \blacksquare

\textbf{Invariant 4.2.4.} Let $s_{\text{before}} \leadsto s_{\text{after}}$ be a reachable transition in the ARS, and $M_{\text{after}}$ be a suitable model for $s_{\text{after}}$. Then, $M_{\text{after}}$ is also suitable for $s_{\text{before}}$.

\textbf{Lemma 4.3.8} Let $s_0 \leadsto \ldots \leadsto s_m$ be a sequence of states produced by an abstract reduction system where $s_0$ is an initial state. Then, Invariant 4.2.4 is preserved by every reachable transition $s_k \leadsto s_{k+1}$ for every $0 \leq k < m$. 

77
4. Soundness of the Reduction from TLA⁺ to SMT

Proof. We prove the lemma by induction on transition steps and similar arguments in the proof of Lemma 4.3.7.

Invariant 4.2.5. Let \( \langle e \mid A \mid \nu \mid \Phi \rangle \) be a reachable state of the ARS, and \( M \) be its extended model. Assume that \( c_{\text{set}} \) is a set cell in the arena \( A \). Then, the following holds:

1. Assume that \( c_{\text{set}} \rightarrow_A c_1, \ldots, c_n \), for some \( n \geq 0 \), and \( c_{\text{set}} \) is introduced by a rule different from (FROMSET). Then, the following holds: \( [c_{\text{set}}]^M \subseteq \{[c_1]^M, \ldots, [c_n]^M\} \).

2. Assume that \( c_{\text{set}} \) is a reduction of the expression FROM \( c_1, \ldots, c_n \) BY \( \theta \) with \( 1 \leq \|\theta\| \leq n \) and \( c_{\text{set}} \rightarrow_A c_{\text{pick}}, \ldots, c_{\text{pick}}^m \). Then, the following holds:

\[ [c_{\text{set}}]^M \subseteq \{[c_{\text{pick}}]^M, \ldots, [c_{\text{pick}}^m]^M\} \]

Lemma 4.3.9 Let \( s_0 \rightarrow \ldots \rightarrow s_m \) be a sequence of states produced by an abstract reduction system where \( s_0 \) is an initial state, and \( s_k = \langle e_k \mid A_k \mid \nu_k \mid \Phi_k \rangle \) for \( 1 \leq k \leq m \). Then, Invariant 4.2.5 is preserved by every state \( s_k \) for every \( 0 \leq k \leq m \).

Proof. We prove the lemma by induction on transition steps. The basis is obviously true because the arena \( A_0 \) is empty. The induction step is proved by case distinction of the applied reduction rules. In the induction step, for every fresh cell \( c_{\text{new}} \) introduced in the state \( s_{k+1} \) such that \( c_{\text{new}} \) has type \( \text{Set}[\tau] \), and \( c_{\text{new}} \rightarrow c_1, \ldots, c_n \), we need to show that \( [c_{\text{new}}]^{M_{k+1}} \subseteq \{[c_1]^{M_{k+1}}, \ldots, [c_n]^{M_{k+1}}\} \). In the following, we show the detailed proof for every case.

- Case (ENUM) \( e_{\text{sub}} = \{c_1, \ldots, c_n\} \):

In this case, only one set cell \( c_{\text{new}} \) is introduced to replace the expression \( e_{\text{sub}} \). By the lemma assumption, we know \( [e_{\text{sub}}]^M_k \subseteq \{[c_1]^M_k, \ldots, [c_n]^M_k\} \). By the rule (ENUM), there is an edge from \( c_{\text{new}} \) to \( c_i \) for every \( 1 \leq i \leq n \), that is, \( c_{\text{new}} \rightarrow c_1, \ldots, c_n \). By Definition 4.1.1, we know \( [e_{\text{sub}}]^{M_{k+1}} = [c_{\text{new}}]^{M_{k+1}} \), and \( [c_i]^{M_k} = [c_i]^{M_{k+1}} \) for every \( i \). Hence, \( [c_{\text{new}}]^{M_{k+1}} \subseteq \{[c_1]^{M_{k+1}}, \ldots, [c_n]^{M_{k+1}}\} \). Moreover, by the rule (ENUM), we have \( c_{\text{new}} \rightarrow c_1, \ldots, c_n \).

- Case (FILTER) \( e_{\text{sub}} = \{x \in c_S : p(x)\} \):

In this case, only one set cell \( c_{\text{new}} \) is introduced to replace the expression \( e_{\text{sub}} \). By the induction hypothesis, we can assume that \( c_S \rightarrow c_1, \ldots, c_n \), and \( [c_S]^M_k \subseteq \{[c_1]^M_k, \ldots, [c_n]^M_k\} \). By the formal semantics of the operator (FILTER) in TLA⁺ (and also in KERA⁺), we know \( [e_{\text{sub}}]^M_k \subseteq [c_S]^M_k \). Hence, we have \( [e_{\text{sub}}]^M_k \subseteq \{[c_1]^M_k, \ldots, [c_n]^M_k\} \). By Definition 4.1.1 it follows \( [c_{\text{new}}]^{M_{k+1}} = [e_{\text{sub}}]^{M_k} \), and \( [c_i]^{M_{k+1}} = [c_i]^{M_k} \) for every \( 1 \leq i \leq n \). It implies that \( \{[c_1]^{M_{k+1}}, \ldots, [c_n]^{M_{k+1}}\} \) is an over-approximation of \( [c_{\text{new}}]^{M_{k+1}} \), i.e. \( [c_{\text{new}}]^{M_{k+1}} \subseteq \{[c_1]^{M_{k+1}}, \ldots, [c_n]^{M_{k+1}}\} \). Moreover, by the rule (FILTER), we have \( c_{\text{new}} \rightarrow c_1, \ldots, c_n \).
– **Case** (DOTDOT) $e_{sub} = c_1..c_2$:

If $c_1, c_2$ are integer constants, the proof is similar to the case (ENUM).

– **Case** (MAP) $e_{sub} = \{ e(x) : x \in S \}$:

Recall that the rule (MAP) works as a “big” step, and contains many “small” steps. Because the “small” steps apply the rules in our reduction system, we can safely assume that Invariant 4.2.5 is preserved in such “small” steps, and only focus on the “big” step. Let $c_{new}$ denote a fresh cell which is introduced to replace the expression $e_{sub}$. We need to show that the appearance of $c_{new}$ would not violate Invariant 4.2.5. By the induction hypothesis, we can assume that $c_S \rightarrow c_1, \ldots, c_n$, and \( \{ c_S \}^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_n \}^{M_k} \} \).

By the rule (MAP), it follows \( \{ e_{sub} \}^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_n \}^{M_k} \} \). By Definition 4.1.1, we have \( \{ c_{new} \}^{M_k+1} = \{ e_{sub} \}^{M_k} \), and \( \{ c_i \}^{M_k+1} = \{ c_i \}^{M_k} \) for every $1 \leq i \leq n$. It implies \( \{ c_{new} \}^{M_k+1} \subseteq \{ \{ c_1 \}^{M_k+1}, \ldots, \{ c_n \}^{M_k+1} \} \). Moreover, by the rule (MAP), we know that $c_{new} \rightarrow c_1, \ldots, c_n$.

– **Case** $e_{sub} = UNION c_S$:

In this case, only one set cell $c_{new}$ is introduced to replace the expression $e_{sub}$. We can assume that $c_S \rightarrow c_1, \ldots, c_n$, and \( \{ c_S \}^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_n \}^{M_k} \} \) by the induction hypothesis. By the formal semantics of the operator UNION in TLA+ (and also in KERAM), we know \( \{ \{ \text{UNION} c_S \} \}^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_n \}^{M_k} \} \).

Moreover, we can assume that $c_S \rightarrow c_1, \ldots, c_{m_i}$, and \( \{ c_S \}^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k} \} \) for every $1 \leq i \leq n$ by the induction hypothesis. We have

\[
\{ c_S \}^{M_k} \cup \ldots \cup \{ c_S \}^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k} \} \cup \ldots \cup \{ \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k} \}
\]

\[
\subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k}, \ldots, \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k} \}
\]

Combining the above results, we have

\[
\text{UNION} c_S^{M_k} \subseteq \{ \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k}, \ldots, \{ c_1 \}^{M_k}, \ldots, \{ c_{m_i} \}^{M_k} \}
\]

By Definition 4.1.1, we have \( \{ c_{new} \}^{M_k+1} = \{ e_{sub} \}^{M_k} \), and \( \{ c_i \}^{M_k+1} = \{ c_i \}^{M_k} \) for every $1 \leq i \leq n$, for every $1 \leq j \leq m_i$. Hence, it follows that

\[
\{ c_{new} \}^{M_k+1} \subseteq \{ \{ c_{m_1} \}^{M_k+1}, \ldots, \{ c_1 \}^{M_k+1}, \ldots, \{ c_{m_i} \}^{M_k+1} \}
\]

By the rule (UNION), there is an edge from $c_{new}$ to $c^i_j$ for every $1 \leq i \leq j$, for every $1 \leq j \leq m_i$. Formally, we have $c_{new} \rightarrow c_1, \ldots, c_{m_i}$.

– **Case** $e_{sub} = \text{CHOOSE} x \in c_S : p$:

Similar to the rule (MAP), the rule (CHOOSE) is a “big” step. Applying similar arguments in the case (MAP), we safely assume that Invariant 4.2.5 is preserved in “small” steps. In
this case, a cell $c_{\text{res}}$ is introduced to replace $c_{\text{sub}}$. If $c_{\text{res}}$ has a basic type, the induction step is clearly correct. Assume that $c_{\text{res}}$ has type $\text{Set}[\tau]$. We only need to show that the cell $c_{\text{res}}$ does not violate Invariant 4.2.5. We do that by proving the case (FROMSET), which is given later.

- **Cases (FUNAPP), (RCDAPP), and (TUPAPP):**

  Similar to the case (CHOOSE), we are reduced to proving these cases by proving the case (FROMSET).

- **Assignment cases $x \in c_S, x \in \text{SUBSET} c_S,$ and $f \in [S \to T]:**

Again, we are reduced to proving these cases by proving the case (FROMSET).

- **Case (FROMSET) FROM $c_1, \ldots, c_n$ BY $\theta$:**

  By the induction hypothesis, we assume $c_i \rightarrow c_i^1, \ldots, c_i^{m_i}$ for every $1 \leq i \leq n$, and $\{[c_i]^{M_k} \subseteq \{[c_i^1]^{M_k}, \ldots, [c_i^{m_i}]^{M_k}\}$. By the rule (FROMSET), we know $m = \max \{m_1, \ldots, m_n\}$, and cells $c_i^{m_i+1}, \ldots, c_i^m$ are exactly the cell $c_i^{m_i}$ for every $i$.

  It follows that $\{[c_i]^{M_k} \subseteq \{[c_i^1]^{M_k}, \ldots, [c_i^{m_i}]^{M_k}\}$. Hence, we have $\{[c_i]^{M_k} \subseteq \{[c_i^1]^{M_k}, \ldots, [c_i^{m_i}]^{M_k}\}$ for every $i$. By Definition 4.1.1 we know $\{[c_i]^{M_k} \subseteq \{[c_i^1]^{M_k}, \ldots, [c_i^{m_i}]^{M_k}\}$ for every $1 \leq i \leq n$ and $1 \leq j \leq l$. It is clear that $\{[c_i]^{M_{k+1}} \subseteq \{[c_i^1]^{M_{k+1}}, \ldots, [c_i^{m_i}]^{M_{k+1}}\}$ for every $i$ (*). By the meaning of the operator (FROMSET) in $\text{KERA}^+$ and the lemma assumption $1 \leq \{\theta\}^{M_{k+1}} \leq n$, we have that

$$\{[c_{\text{pick}}]^{M_{k+1}} = \{[\theta]^{M_{k+1}} \subseteq \text{FROM} c_1, \ldots, c_n \text{ BY } \theta\}^{M_{k+1}} \subseteq \text{FROM} c_1, \ldots, c_n \text{ BY } \theta\}^{M_{k+1}}$$

for every $1 \leq j \leq m$. Applying (*) to $c_\theta$, it follows $\{[c_\theta]^{M_{k+1}} \subseteq \{[c_1]^{M_{k+1}}, \ldots, [c_m]^{M_{k+1}}\}$. Hence, $\{[c_{\text{pick}}]^{M_{k+1}} \subseteq \{[c_1^1]^{M_{k+1}}, \ldots, [c_m^m]^{M_{k+1}}\}$. By the rule (FROMSET), it is easy to check that $c_{\text{pick}} \rightarrow c_1^1, \ldots, c_m^m$.

- **Other cases:** In these cases, the reduction rule does not introduce a fresh set cell. Therefore, the induction step is obviously true.

Finally, by combining the above results, we know that Invariant 4.2.5 is preserved by every state $s_k$ for every $0 \leq k \leq m$.

**Invariant 4.2.6.** Let $\langle e | A | \nu | \Phi \rangle$ be a reachable state of the ARS. Assume that $c_f$ is a function cell of type $\tau_1 \rightarrow \tau_2$ in the arena $A$. Then, there is a cell $c_{\text{rel}}$ of type $\text{Set}[\tau_1 \ast \tau_2]$ such that the function cell is pointing to it: $c_f \rightarrow_A c_{\text{rel}}$.

**Lemma 4.3.10** Let $s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_m$ be a sequence of states produced by an abstract reduction system where $s_0$ is an initial state, and $s_k = \langle e_k | A_k | \nu_k | \Phi_k \rangle$ for $1 \leq k \leq m$. Then, Invariant 4.2.6 is preserved by every state $s_k$ for every $0 \leq k \leq m$. 

80
4.3. Detailed proofs

Proof. We prove the lemma by induction on transition steps. The basis is obviously correct because the arena $A_0$ is empty.

The induction step is proved by case distinction on the applied reduction rules. Because no rule removes anything from the arena, we only need to focus on when a fresh function cell is introduced. Notice that our rewriting system has only few rules related to functions which are (FUNCTOR), (FUNEXC), (DOMAIN), (FUNAPP), and (PICKFUN). The other rules do not introduce a fresh function cell. Moreover, the rule (DOMAIN) never introduce a function cell. Therefore, we only need to consider four cases (FUNCTOR), (FUNEXC), (FUNAPP), and (PICKFUN). We here show the proof for only two cases (FUNCTOR) and (FUNEXC). Similary arguments apply to the cases (FROMFUNC) and (FUNAPP).

Whenever a function cell $c_f$ with a type $\tau_1 \to \tau_2$ is introduced by the rules (FUNCTOR) a d (FUNEXC), it is easy to check that a corresponding set cell $c_{rel}$ with a type $\text{Set}(\tau_1 * \tau_2)$ is also introduced, and they are connected, i.e. $c_f \Downarrow c_{rel}^f$. Therefore, the induction step is correct.

The proof of Invariant 4.2.7 requires following Lemmas 4.3.11 and 4.3.12.

Lemma 4.3.11 Let $s_{before} \rightarrow s_{after}$ be a reachable transition where the reduction rule (FUNCTOR) is applied to replace a sub-expression $[x \in c_S \mapsto e]$ in $e_{before}$ with a function cell $c_f$, and to produce the expression $c_{after}$. Assume that $c_f \Downarrow A_{after} c_{rel}^f$. Let $T$ denote a set $\{ \langle x, f(x) \rangle : x \in \text{DOMAIN } f \}$. Let $M_{before}, M_{after}$ be extended models of $s_{before}, s_{after}$, respectively. Then, the following holds: $[T]^{M_{after}} = [c_{rel}^f]^{M_{after}}$.

Proof. Because $s_{before} \rightarrow s_{after}$ is a reachable state, Invariants 4.2.1, 4.2.6 are preserved by states $s_{before}, s_{after}$. By the formal semantics of the function constructor (FUNCTOR) $\{ x \in S \mapsto e \}$ in TLA$, it follows that $[\text{DOMAIN } f]^{M_{after}} = [c_S]^{M_{after}}$. Assume that a function cell $c_f$ has type $\tau_1 \to \tau_2$ for some types $\tau_1, \tau_2$. By the definition of $c_{rel}^f$ and $T$, both $c_{rel}^f$ and $T$ have the same type $\text{Set}(\tau_1 * \tau_2)$. The proof of Lemma 4.3.11 falls naturally into two steps.

Case (a). If $c_S$ is a syntactically empty set $\emptyset$, both $[T]^{M_{after}}$ and $[c_{rel}^f]^{M_{after}}$ are empty sets. Therefore, the lemma statement is obviously correct in this case.

Case (b): $c_S$ is not statically empty, that is, $c_S \to c_1, \ldots, c_n$ for some $n > 0$. We prove set equality by considering two cases.

Case (b.1): $[T]^{M_{after}} \subseteq [c_{rel}^f]^{M_{after}}$

Let $\lfloor \langle a, f(a) \rangle \rfloor^{M_{after}}$ be an arbitrary member of $[T]^{M_{after}}$, i.e. $\lfloor \langle a, f(a) \rangle \rfloor^{M_{after}} \in [T]^{M_{after}}$. By the construction of $T$, we know that $\lfloor a \rfloor^{M_{after}} \in [\text{DOMAIN } f]^{M_{after}}$. Therefore, $\lfloor a \rfloor^{M_{after}}$ is also a member of $[c_S]^{M_{after}}$ since $\lfloor c_S \rfloor^{M_{after}} = [\text{DOMAIN } f]^{M_{after}}$. Moreover, by the formal semantics of the function constructor in TLA$, we have $\lfloor f(a) \rfloor^{M_{after}} = \lfloor e(a) \rfloor^{M_{after}}$.

By Invariant 4.2.5, it follows that $\lfloor c_S \rfloor^{M_{after}} \subseteq \lfloor \{ c_1, \ldots, c_n \} \rfloor^{M_{after}}$. Since $\lfloor a \rfloor^{M_{after}} \in [c_S]^{M_{after}}$, there must be a cell $c_i$ such that $\lfloor c_i \rfloor^{M_{after}} = \lfloor a \rfloor^{M_{after}}$ and $1 \leq i \leq n (**)$.
Therefore, $\llbracket c_i \rrbracket^{M_{after}}$ is also a member of both $\llbracket c_j \rrbracket^{M_{after}}$ and $\llbracket \text{domain } f \rrbracket^{M_{after}}$. By Lemma 4.3.4 it follows that $\llbracket e(a) \rrbracket^{M_{after}} = \llbracket e(c_i) \rrbracket^{M_{after}}$.

By the definition of the rule (FUNCTOR), we know that this rule contains “small” steps to rewrite expressions $e(c_1), \ldots, e(c_n)$. Therefore, the translation must contain a sequence of states from $s_i = (e(c_1), \nu_1, A_1, \Phi_i)$ to $s_i^j = (c_i^j, \nu_1^j, A_1^j, \Phi_i^j)$, i.e. $s_i \leadsto s_i^1 \leadsto \ldots \leadsto s_i^n$. Recall that $M_{after}$ is an extension of $M_i$. By Definition 4.1.1 and Lemma 4.3.4, we have $\llbracket e(c_i) \rrbracket^{M_{after}} = \llbracket c_i^j \rrbracket^{M_{after}}$. So, we have $\llbracket c_i^j \rrbracket^{M_{after}} = \llbracket e(a) \rrbracket^{M_{after}} = \llbracket f(a) \rrbracket^{M_{after}}$.

Now, we have $\llbracket (a, f(a)) \rrbracket^{M_{after}} = \llbracket (c_i, c_i^j) \rrbracket^{M_{after}}$. By (\ast), we already know $\llbracket c_i \rrbracket^{M_{after}} \in \llbracket c_j \rrbracket^{M_{after}}$. By the definition of $c_{rel}^f$, we have $\llbracket (c_i, c_i^j) \rrbracket^{M_{after}} \in \llbracket c_{rel}^f \rrbracket^{M_{after}}$. So, we have that $\llbracket (a, f(a)) \rrbracket^{M_{after}} \in \llbracket c_{rel}^f \rrbracket^{M_{after}}$. Since $\llbracket (a, f(a)) \rrbracket^{M_{after}}$ is an arbitrary member of $\llbracket T \rrbracket^{M_{after}}$, we have $\llbracket T \rrbracket^{M_{after}} \subseteq \llbracket c_{rel}^f \rrbracket^{M_{after}}$.

**Case (b,2):** $\llbracket c_{rel}^f \rrbracket^{M_{after}} \subseteq \llbracket T \rrbracket^{M_{after}}$

Let $\llbracket (a, \text{res}) \rrbracket^{M_{after}}$ be an arbitrary member of $\llbracket c_{rel}^f \rrbracket^{M_{after}}$, i.e. $\llbracket (a, \text{res}) \rrbracket^{M_{after}} \subseteq \llbracket c_{rel}^f \rrbracket^{M_{after}}$. By the definition of $c_{rel}^f$, we know that $\llbracket a \rrbracket^{M_{after}} \in \llbracket c_i \rrbracket^{M_{after}}$. By the definition of the rule (FUNCTOR), we know that this rule contains small rewriting steps related to expressions $e(c_1), \ldots, e(c_n)$. Therefore, the translation must contain a sequence of states from $s_i = (e(c_1), \nu_1, A_1, \Phi_i)$ to $s_i^1 = (c_i^1, \nu_1^1, A_1^1, \Phi_i^1)$, i.e. $s_i \leadsto s_i^1 \leadsto \ldots \leadsto s_i^n$ for every $1 \leq i \leq n$. By Invariant 4.2.5, we know that $\llbracket c_{rel}^f \rrbracket^{M_{after}} \subseteq \llbracket (c_1, c_1^1), \ldots, (c_n, c_n^1) \rrbracket^{M_{after}}$. Therefore, there must be an integer $k$ such that $1 \leq k \leq n$, and $\llbracket (a, \text{res}) \rrbracket^{M_{after}} = \llbracket (c_k, c_k^1) \rrbracket^{M_{after}}$. It implies that $\llbracket a \rrbracket^{M_{after}} = \llbracket c_k \rrbracket^{M_{after}}$. Hence, we have $\llbracket \text{res} \rrbracket^{M_{after}} = \llbracket c_k \rrbracket^{M_{after}}$, $\llbracket e(a) \rrbracket^{M_{after}} = \llbracket e(c_k) \rrbracket^{M_{after}}$.

Recall that we already know $\llbracket c_j \rrbracket^{M_{after}} = \llbracket \text{domain } f \rrbracket^{M_{after}}$. Therefore, we have $\llbracket a \rrbracket^{M_{after}}$ is a member of both $\llbracket c_j \rrbracket^{M_{after}}$ and $\llbracket \text{domain } f \rrbracket^{M_{after}}$. Now by the formal semantics of the constructor (FUNCTOR) in TLA$^+$, we know that $\llbracket f(a) \rrbracket^{M_{after}} = \llbracket e(a) \rrbracket^{M_{after}} = \llbracket \text{res} \rrbracket^{M_{after}}$. By the construction of $T$, we have $\llbracket (a, \text{res}) \rrbracket^{M_{after}} \subseteq \llbracket T \rrbracket^{M_{after}}$. Since $\llbracket (a, \text{res}) \rrbracket^{M_{after}}$ be an arbitrary member of $\llbracket c_{rel}^f \rrbracket^{M_{after}}$, we have $\llbracket c_{rel}^f \rrbracket^{M_{after}} \subseteq \llbracket T \rrbracket^{M_{after}}$.

**Lemma 4.3.12** Let $s_{before} \rightarrow s_{after}$ be a reachable transition where the reduction rule (FUNEXC) is applied to replace a sub-expression $g \triangleq [f \text{ except! } [a = r]]$ in $e_{before}$ with a function cell $c_g$, and to produce the expression $e_{after}$. Assume that $c_g \rightarrow A_{after} c_{rel}^f$. Let $T$ denote a set $\{ \langle x, g(x) \rangle : x \in \text{domain } g \}$, and $M_{before}, M_{after}$ be extended models of $s_{before}, s_{after}$, respectively. Then, the following holds: $\llbracket T \rrbracket^{M_{after}} = \llbracket c_{rel}^f \rrbracket^{M_{after}}$.

**Proof.** Because $s_{before} \rightarrow s_{after}$ is a reachable state, Invariants 4.2.4,4.2.6 are preserved by states $s_{before}, s_{after}$. Recall that we have $\llbracket \text{domain } g \rrbracket^{M_{after}} \equiv \llbracket \text{domain } f \rrbracket^{M_{after}}$ by the formal semantics of the constructor (EXCEPT) in TLA$^+$. If $\llbracket a \rrbracket^{M_{after}}$ is not in the domain of $\llbracket g \rrbracket^{M_{after}}$, we know that $\llbracket g \rrbracket^{M_{after}} = \llbracket f \rrbracket^{M_{after}}$. Moreover, by the construct of $c_g$, we have $\llbracket c_{rel}^g \rrbracket^{M_{after}} = \llbracket c_{rel}^f \rrbracket^{M_{after}}$. In this case, the statement is obviously correct. In the following, we consider case $\llbracket a \rrbracket^{M_{after}} \in \llbracket \text{domain } g \rrbracket^{M_{after}}$. Notice that $f$ can be defined with the constructor (FUNCTOR) or the constructor (EXCEPT). Here we show only the
proof for case where \( f \) is defined with the constructor (functor). The proof for the case (except) is similar since we need to use only the fact \( \llbracket \{ x, f(x) : x \in \text{domain } f \} \rrbracket^\mathcal{M}_{\text{after}} = \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} \). The following proof in which \( f \) is defined by a function definition contains two cases.

- **Case** \( \llbracket T \rrbracket^\mathcal{M}_{\text{after}} \subseteq \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} : \) Let \( \llbracket \langle \text{arg}, g(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \) be an arbitrary member of \( \llbracket T \rrbracket^\mathcal{M}_{\text{after}} \), i.e., \( \llbracket \langle \text{arg}, g(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \in \llbracket T \rrbracket^\mathcal{M}_{\text{after}} \). By Lemma 4.3.12 we know that \( \llbracket \langle \text{arg}, f(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \in \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} \).

If \( \llbracket \text{arg} \rrbracket^\mathcal{M}_{\text{after}} \neq \llbracket a \rrbracket^\mathcal{M}_{\text{after}} \), we know that \( \llbracket f(\text{arg}) \rrbracket^\mathcal{M}_{\text{after}} = \llbracket g(\text{arg}) \rrbracket^\mathcal{M}_{\text{after}} \). Therefore, by the construction of \( c^f_r \), we know \( \llbracket \langle \text{arg}, g(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \subseteq \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} \).

Now consider case \( \llbracket a \rrbracket^\mathcal{M}_{\text{after}} = \llbracket \text{arg} \rrbracket^\mathcal{M}_{\text{after}} \). By the formal semantics of the constructor (except) in TLA\(^+\) (and also in KERA\(^+\)), we know that \( \llbracket g(\text{arg}) \rrbracket^\mathcal{M}_{\text{after}} = \llbracket r \rrbracket^\mathcal{M}_{\text{after}} \).

Therefore, we have \( \llbracket \langle a, r \rangle \rrbracket^\mathcal{M}_{\text{after}} = \llbracket \langle \text{arg}, g(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \). Moreover, we have

\[
\llbracket \text{ite}(p[1] = a, \langle a, r \rangle, p) \rrbracket^\mathcal{M}_{\text{after}} = \llbracket \langle a, r \rangle \rrbracket^\mathcal{M}_{\text{after}}
\]

By the construction of \( c^f_r \), we have that \( \llbracket \langle \text{arg}, g(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \subseteq \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} \).

- **Case** \( \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} \subseteq \llbracket g \rrbracket^\mathcal{M}_{\text{after}} : \) Let \( \llbracket \langle \text{arg}, \text{res} \rangle \rrbracket^\mathcal{M}_{\text{after}} \) be an arbitrary member of \( \llbracket c^f_r \rrbracket^\mathcal{M}_{\text{after}} \).

If \( \llbracket \text{arg} \rrbracket^\mathcal{M}_{\text{after}} \neq \llbracket a \rrbracket^\mathcal{M}_{\text{after}} \), we know \( \llbracket \langle \text{arg}, g(\text{arg}) \rangle \rrbracket^\mathcal{M}_{\text{after}} \in \llbracket T \rrbracket^\mathcal{M}_{\text{after}} \) by similar arguments in the previous case. Now consider case \( \llbracket a \rrbracket^\mathcal{M}_{\text{after}} = \llbracket \text{arg} \rrbracket^\mathcal{M}_{\text{after}} \). By the construction of \( c^f_r \), we know that

\[
\llbracket \langle \text{arg}, \text{res} \rangle \rrbracket^\mathcal{M}_{\text{after}} = \llbracket \langle a, r \rangle \rrbracket^\mathcal{M}_{\text{after}}
\]

Moreover, the formal semantics of the constructor (except) in TLA\(^+\) (and also in KERA\(^+\)), we know that \( \llbracket g(\text{arg}) \rrbracket^\mathcal{M}_{\text{after}} = \llbracket r \rrbracket^\mathcal{M}_{\text{after}} \). It implies \( \llbracket \langle \text{arg}, \text{res} \rangle \rrbracket^\mathcal{M}_{\text{after}} \subseteq \llbracket g \rrbracket^\mathcal{M}_{\text{after}} \).

\[
\text{Invariant 4.2.7.} \quad \text{Let } \langle c \mid A \mid \nu \mid \Phi \rangle \text{ be a reachable state of the ARS, and } \mathcal{M} \text{ be its extended model. Assume that } c_f \text{ is a function cell, and } c_f \to_A c_r. \text{ Then, it follows that the set } \llbracket c_r \rrbracket^\mathcal{M}_{\text{after}} \text{ is equal to the set } \llbracket \{ x, f(x) : x \in \text{domain } f \} \rrbracket^\mathcal{M}_{\text{after}}.
\]

**Lemma 4.3.13** Let \( s_0 \leadsto \ldots \leadsto s_m \) be a sequence of states produced by an abstract reduction system where \( s_0 \) is an initial state, and \( s_k = \langle e_k \mid A_k \mid v_k \mid \Phi_k \rangle \) for \( 1 \leq k \leq m \). Then, Invariant 4.2.7 is preserved by every state \( s_k \) for every \( 0 \leq k \leq m \).

**Proof.** We prove the lemma by induction on transition steps. The basis is obviously correct because the arena \( A_0 \) is empty. The induction step is proved case distinction on the applied reduction rules. Because no rule removes anything from the arena, we only need to focus on when a fresh function cell is introduced. Therefore, we only need to consider four cases (functor), (funexec), (funapp), and (pickfun) because other rules do not introduce a fresh function cell. We here show the proof for only two cases (functor) and (funexec). Similar arguments apply to the cases (fromfunc) and (funapp). The case (functor) is correct by Lemma 4.3.11 and the case (funapp) is correct by Lemma 4.3.12 Therefore, the lemma is correct.
4. Soundness of the Reduction from TLA$^+$ to SMT

**Proposition 4.2.2.** Let $s_0 \leadsto \ldots \leadsto s_m$ be a sequence of states produced by an abstract reduction system. Then, Invariants 4.2.3 and 4.2.4 are preserved by every transition $s_i \leadsto s_{i+1}$ for every $0 \leq s < m$. Moreover, Invariants 4.2.1, 4.2.2, and 4.2.5–4.2.7 are preserved by every state $s_j$ for every $0 \leq j \leq m$.

**Proof.** We prove the proposition by induction on transition steps. The basis is obviously correct. The induction step is correct by Lemmas 4.3.5–4.3.13. Therefore, Proposition 4.2.2 is correct.

### 4.3.5 Soundness

The proof of Theorem 4.2.2 requires Lemmas 4.3.14–4.3.20.

**Lemma 4.3.14** Let $s_{\text{before}} \leadsto s_{\text{after}}$ be a reachable transition where the applied reduction rule is one of the rules in Section 3.6. Assume that the reduction rule replaces a sub-expression $e_{\text{sub}}$ in a formula $e_{\text{before}}$ with a fresh cell $c_{\text{new}}$. Moreover, assume that the formula $e_{\text{before}} \land \Phi_{\text{before}}$ is satisfiable. Let $M_{\text{before}}$ be a model of $e_{\text{before}} \land \Phi_{\text{before}}$, and $M_{\text{after}}$ be an extended model from $M_{\text{before}}$ as in Definition 4.1.1. Then, $M_{\text{after}}$ is a model of both $e_{\text{after}} \land \Phi_{\text{after}}$.

**Proof.** Here we assume that $e_{\text{sub}}$ has type $\text{Set}[\tau]$, and thus, its elements have type $\tau$ in the following proof. Recall that $\llbracket e_{\text{sub}} \rrbracket^M_{\text{after}} = \llbracket c_{\text{new}} \rrbracket^M_{\text{after}}$ by Definition Definition 4.1.1. By Lemma 4.3.4 it is easily seen that $M_{\text{after}} \models e_{\text{after}}$. Now we prove that $M_{\text{after}} \models \Phi_{\text{after}}$. The proof is done by case distinction on the applied reduction rules. We first outline the strategy to prove each of the cases in the following.

1. By Definition 4.1.1 we know $M_{\text{after}} \models e_{\text{before}} \land \Phi_{\text{before}}$, and $M_{\text{after}} \models c_{\text{new}} = e_{\text{sub}}$. By Lemma 4.3.4 it follows $M_{\text{after}} \models e_{\text{after}}$.

2. Let $\text{target} \triangleq c_{\text{new}} = e_{\text{sub}}$. Our goal is to prove that $\text{target}$ implies $\Delta$ where $\Delta$ is a fresh constraint in $\Phi_{\text{after}}$, i.e. $\Phi_{\text{after}} = \Phi_{\text{before}} \land \Delta$.

3. We rewrite the expression $\text{target}$ by considering the formal semantics of an operator to construct $\text{target}$, and the arena $A_{\text{after}}$ and Invariant 4.2.5. Assume that the rewriting step produces a new formula $\text{res}_1$. If $\text{res}_1$ contains an operator which does not have a counterpart in SMT, we reset $\text{target} \triangleq \text{res}_1$, and repeat this step again with the new target. Otherwise, we go to the next step.

4. Prove that $\text{target}$ implies $\Delta$ under $M_{\text{after}}$.

5. It follows $M_{\text{after}} \models \Delta$, and thus, $M_{\text{after}} \models \Phi_{\text{after}}$.

6. By (1) and (5), we have $M_{\text{after}} \models e_{\text{after}} \land \Phi_{\text{after}}$.

We now turn to the detailed proof of the cases in Lemma 4.3.14.
- Case $\textit{ENUM} \ e_{\text{sub}} = \{c_1, \ldots, c_n\}$:

Our initial target is $\llbracket c_{\text{new}} \rrbracket_M^{k+1} = \llbracket e_{\text{sub}} \rrbracket_M^{k+1}$. By the definition of $e_{\text{sub}}$, we have

$$\llbracket c_{\text{new}} \rrbracket_M^{k+1} = \llbracket e_{\text{sub}} \rrbracket_M^{k+1} \iff \llbracket c_{\text{new}} \rrbracket_M^{k+1} = \left\{ \llbracket c_1 \rrbracket_M^{k+1}, \ldots, \llbracket c_n \rrbracket_M^{k+1} \right\}$$

By Invariant 4.2.5, it follows $\left\{ \llbracket c_1 \rrbracket_M^\text{after}, \ldots, \llbracket c_n \rrbracket_M^\text{after} \right\}$ is an over-approximation of $\llbracket c_{\text{new}} \rrbracket_M^\text{after}$. Therefore, we can apply Lemma 4.3.3, and rewrite the above formula as follows. The last formula is obtained by applying Invariant 4.2.2, and replacing every set membership with a corresponding propositional symbol.

$$\llbracket c_{\text{new}} \rrbracket_M^\text{after} = \left\{ \llbracket c_1 \rrbracket_M^\text{after}, \ldots, \llbracket c_n \rrbracket_M^\text{after} \right\}$$

$$\iff \bigwedge_{i=1}^n \llbracket c_i \rrbracket_M^\text{after} \in \llbracket c_{\text{new}} \rrbracket_M^\text{after} \iff \llbracket c_i \rrbracket_M^\text{after} \in \left\{ \llbracket c_1 \rrbracket_M^\text{after}, \ldots, \llbracket c_n \rrbracket_M^\text{after} \right\}$$

- Case $\textit{FILTER} \ e_{\text{sub}} = \{x \in c_S : p(x)\}$:

If $c_S$ is statically an empty set, there is no edge from $c_S$ in arena $A_{\text{after}}$. It results in no constraint in (INFILTER). Therefore, $M_{\text{after}}$ is a model of $\Phi_{\text{after}}$ because constraints $\Phi_{\text{after}}$ and $\Phi_{\text{before}}$ are the same. We now turn to the case where $c_S$ is not statically an empty set. Assume that $c_S \rightarrow c_1, \ldots, c_n$ in the arena $A_{\text{after}}$. Recall that the first step in the rule (FILTER) sequentially rewrites $p(c_i/x)$ into a cell $c_i^\text{after}$ for every $1 \leq i \leq n$, which can be considered as “small” steps. Since these steps also apply reduction rules in our rewriting systems, here we can assume that $M_{\text{after}}$ is already a model of constraints produced in such steps. By Invariant 4.2.5, we know that a finite set $\left\{ \llbracket c_1 \rrbracket_M^\text{after}, \ldots, \llbracket c_n \rrbracket_M^\text{after} \right\}$ is an over-approximation of $\llbracket c_{\text{new}} \rrbracket_M^\text{after}$. By Lemma 4.3.3, we have

$$\llbracket c_{\text{new}} \rrbracket_M^\text{after} = \llbracket e_{\text{sub}} \rrbracket_M^\text{after}$$

$$\iff \llbracket c_{\text{new}} \rrbracket_M^\text{after} = \left\{ \{x \in c_S : p(x)\} \right\} \llbracket \text{after} \right\}$$

$$\iff \bigwedge_{c \in \{c_1, \ldots, c_n\}} (\llbracket c \rrbracket_M^\text{after} \in \llbracket c_{\text{new}} \rrbracket_M^\text{after} \iff \llbracket c \rrbracket_M^\text{after} \in \left\{ \{x \in c_S : p(x)\} \right\} \llbracket \text{after} \right\})$$

$$\iff \bigwedge_{c \in \{c_1, \ldots, c_n\}} (\llbracket c \rrbracket_M^\text{after} \in \llbracket c_{\text{new}} \rrbracket_M^\text{after} \iff \left( \llbracket c \rrbracket_M^\text{after} \in \llbracket c_S \rrbracket_M^\text{after} \land \llbracket p(c) \rrbracket_M^\text{after} \right)$$

$$\iff \bigwedge_{i=1}^n (\llbracket c_i \rrbracket_M^\text{after} \in \llbracket c_{\text{new}} \rrbracket_M^\text{after} \iff \left( \llbracket c_i \rrbracket_M^\text{after} \in \llbracket c_S \rrbracket_M^\text{after} \land \llbracket p(c_i) \rrbracket_M^\text{after} \right)$$

85
4. Soundness of the Reduction from TLA\textsuperscript{+} to SMT

By Invariant [4.2.2] we safely replace the operator $\in$ with the propositional symbol $en$ for the edges. So, we have $\forall i. (\forall c. (c, i) \in e[i]) \leftrightarrow (\forall c. (c, i) \in e[i] \land (c, c_i) \in e[i])$ for every $1 \leq i \leq n$. It is the constraint of (infilter), and therefore, $\mathcal{M}_{after} \models \Phi_{after}$. By Lemma [4.3.4] and the lemma assumption, we know $\mathcal{M}_{after} \models e_{after}$. Hence, it follows $\mathcal{M}_{after} \models e_{after} \land \Phi_{after}$.

- **Case** $\text{MAP } e_{sub} = \{ e(x) : x \in S \}$:

If $c_S$ is statically the empty set, the proof is similar to the previous case (filter). Now assume that $c_S$ is not statically the empty set, and $c_1, \ldots, c_n$ cells connected to $c_S$ in arena $A_{after}$. Notice that the rule (map) is a “big” step, and contains “small” steps. Because such “small” steps also use the rules in our reduction system, we can safely assume that $\mathcal{M}_{after}$ is already a model of constraints generated in during the translation $e[c_i/x] \Rightarrow c^e$. By Invariant [4.2.5], we know that a finite set $\{ [c_1]_{after}, \ldots, [c_n]_{after} \}$ is an over-approximation of $[c]_{after}$. Moreover, a finite set $\{ [c_1]_{after}, \ldots, [c_n]_{after} \}$ is an over-approximation of both $[c_{sub}]_{after}$ and $[e_{sub}]_{after}$. By Lemma [4.3.2], we have

$\mathcal{M}_{after} := e_{sub} \models [c_{new}]_{after} \leftrightarrow [c_{new}]_{after} \models [e_{sub}]_{after} \Rightarrow [c_{new}]_{after} \models [\{ e(x) : x \in c_S \}]_{after}$

$\Rightarrow \bigwedge_{i=1}^{n} \left( [c^e_i]_{after} \in [c_{new}]_{after} \Rightarrow [c^e_i]_{after} \models [\{ e(x) : x \in c_S \}]_{after} \right)$

$\Rightarrow \bigwedge_{i=1}^{n} \left( [c^e_i]_{after} \in [c_{new}]_{after} \Rightarrow \bigvee_{j=1}^{n} \left( [c^e_i]_{after} = [c^e_j]_{after} \land [c^e_j]_{after} \in [c_S]_{after} \right) \right)$

Because both $c^e_i, c^e_j$ are from $c^e_1, \ldots, c^e_n$, instead of the big disjunction in the last formula, we can consider the case where these index $i, j$ are the same, i.e. $i = j$. Moreover, we can replace the equality comparison in such formula by $i = j$. Therefore, for every $1 \leq i \leq n$, we have

$[c^e_i]_{after} \in [c_{new}]_{after} \Rightarrow [c^e_i]_{after} \models [c^e_i]_{after} \models [c_S]_{after}$

$\Rightarrow \left( [en(c_{new}, i, c^e_i)]_{after} \leftrightarrow [en(c_S, i, c_i)]_{after} \right)$

(By Invariant [4.2.2])

Hence, it follows It is the constraint (inmap), and therefore, the induction step is correct.

- **Case** $\text{DOTDOT } e_{sub} = c_1 \ldots c_2$:

Similar to the case (enum) because our requirements on the use of the operator DOTDOT.

- **Case** $e_{sub} = c_{sub} \in \text{SUBSET } c_S$:

If $c_S$ is statically an empty set $\emptyset$, similar arguments in the case (filter) is applied. Now assume that $c_S$ is not statically an empty set, and $c_1, \ldots, c_n$. By Invariant [4.2.5], we know that a finite set $\{ [c_1]_{after}, \ldots, [c_n]_{after} \}$ is an over-approximation of both $[c_{sub}]_{after}$ and $[e_{sub}]_{after}$. By Lemma [4.3.2], we have

$\mathcal{M}_{after} \models [c_{sub}]_{after} \Leftrightarrow [e_{sub}]_{after}$

86
We prove only the most interesting case in which both $c_S$ and its member are not tactically the empty set. For other cases, the proof is similar but much simpler. Assume that $c_1^S, \ldots, c_n^S$ are cells connected to $c_S$, and that $c_1^m, \ldots, c_m^m$ are cells connected to $c_i^S$ in such arena $A_{after}$ for every $i$. By Invariant 4.2.5, we know that the finite set $\{[c_1^s]_{after}, \ldots, [c_n^s]_{after}\}$ is an over-approximation of $[c_S]_{after}$. Moreover, by the definition of $c_{set}$ and $\text{UNION}$, we know that the merge of such sets $\{[c_1^s]_{after}, \ldots, [c_m^s]_{after}\}$ is an over-approximation of both $[c_{set}]_{after}$ and $[\text{UNION} c_S]_{after}$. By Lemma 4.3.2, we have

\[
\begin{align*}
&\iff [c_{set}]_{after} = [c_{sub}]_{after} \\
&\iff [c_{set}]_{after} = [\text{UNION} c_S]_{after} \\
&\iff \left( \bigwedge_{c \in \{c_1^s, \ldots, c_n^s\}} [c]_{after} \in [c_{set}]_{after} \iff [c]_{after} \in [\text{UNION} c_S]_{after} \right) \\
&\iff \left( \bigwedge_{c \in \{c_1^s, \ldots, c_m^s\}} [c]_{after} \in [c_{set}]_{after} \iff [c]_{after} \in [\text{UNION} c_S]_{after} \right)
\end{align*}
\]

Again, by Invariant 4.2.5, we know that $\{[c_1^s]_{after}, \ldots, [c_n^s]_{after}\}$ is an over-approximation of $[c_S]_{after}$. Therefore, for every $c \in \{c_1^s, \ldots, c_n^s\}$, we have

\[
[c]_{after} \in [\text{UNION} c_S]_{after} \iff \left( \bigvee_{i=1}^n [c_i^s]_{after} \in [c_S]_{after} \land [c]_{after} \in [c_i^s]_{after} \right).
\]

By the arena structure, instead of the big disjunction on the above formula, if $c = c_j^i$ for some $1 \leq i \leq n$, some $1 \leq j \leq m_i$, then we can consider a similar constraint but only with $c_j^i$. Formally, for every $1 \leq i \leq n$, for every $1 \leq j \leq m_i$, we have

\[
[c_j^i]_{after} \in [\text{UNION} c_S]_{after} \iff \left( [c_j^i]_{after} \in [c_S]_{after} \land [c]_{after} \in [c_j^i]_{after} \right).
\]

By Invariant 4.2.2, we safely replace the operator $\in$ with the predicate $en$. So, for every $1 \leq i \leq n$, for every $1 \leq j \leq m_i$, we have

\[
[en(c_{set}, idx_{i,j}, c_j^i)]_{after} \iff \left( [en(c_S, j, c_j^i)]_{after} \land [en(c_S, i, c_j^i)]_{after} \right)
\]
4. Soundness of the Reduction from TLA⁺ to SMT

It means $\mathcal{M}_{after}$ is a model of constraint $InU$. Therefore, the induction step is correct

- **Case (SETEQ) $c_S = c_T$:**

By similar arguments in the case (MAP), we can assume that $\mathcal{M}_{after}$ is a model of constraints produced in “small” steps. By Definition 4.1.1 and the axiom of set equality, we have

$$\llbracket c_{eq} \rrbracket^\mathcal{M}_{after} \iff (\forall x \in S \cdot x \in T)^\mathcal{M}_{after} \land (\forall y \in T \cdot y \in S)^\mathcal{M}_{after} \iff \llbracket c_S = c_T \rrbracket^\mathcal{M}_{after}$$

- **Case (SETCARD):**

The constraints produced in the rule (SETCARD) by our reduction system is the naive method to decide the cardinality of a set.

- **Cases $e_{sub} = x \in c_S$, or $e_{sub} = c \in SUBSET c_S$ or $e_{sub} = c \in [c_{S1} \rightarrow c_{S2}]:**

Similar to the case (ASGN) $x \in c_S$ in Lemma 4.3.16 which is proved later.

**Lemma 4.3.15** Let $s_{before} \leadsto s_{after}$ be a reachable transition where the applied reduction rule is one of rules in Section 3.7. Assume that the formula $e_{before} \land \Phi_{before}$ is satisfiable. Let $\mathcal{M}_{before}$ be a model of $e_{before} \land \Phi_{before}$ such that $1 \leq \llbracket \theta \rrbracket^\mathcal{M}_{before} \leq n$, and $\mathcal{M}_{after}$ be an extended model from $\mathcal{M}_{before}$ as in Definition 4.1.1. Then, $\mathcal{M}_{after}$ is a model of both $e_{after} \land \Phi_{after}$.

**Proof.** Because $s_{before} \leadsto s_{after}$ is a reachable transition, Invariants 4.2.1-4.2.7 are preserved by states $s_{before}, s_{after}$. The proof of Lemma 4.3.15 is done by case distinct of the applied reduction rule.

- **Case (FROMBASIC) FROM $c_1, \ldots, c_n$ BY $\theta$:**

By Lemma 4.3.4 and Definition 4.1.1, it is easily seen that $\mathcal{M}_{after} \models e_{after}$. Now we prove that $\mathcal{M}_{after} \models \Phi_{after}$. By the fact $1 \leq \llbracket \theta \rrbracket^\mathcal{M}_{before} \leq n$, there is an $i_0$ such that $1 \leq i_0 \leq n$ and $\theta = \theta_{i_0}$. By the formal semantics of the operator (FROMBASIC) and the fact $1 \leq \llbracket \theta \rrbracket^\mathcal{M}_{before} \leq n$, we know that $\llbracket \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \rrbracket^\mathcal{M}_{after} = \llbracket c_{i_0} \rrbracket^\mathcal{M}_{after}$. By Definition 4.1.1 and the fact $1 \leq \llbracket \theta \rrbracket^\mathcal{M}_{before} \leq n$, we know $\llbracket c_{\text{pick}} \rrbracket^\mathcal{M}_{after} = \llbracket \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \rrbracket^\mathcal{M}_{after}$.

Combining the above results, we have $\llbracket c_{\text{pick}} \rrbracket^\mathcal{M}_{after} = \llbracket c_{i_0} \rrbracket^\mathcal{M}_{after}$. Because $1 \leq i_0 \leq n$, it immediately follows that $\mathcal{M}_{after} \models \bigwedge_{1 \leq i \leq n} (\theta = i \Rightarrow c_{\text{pick}} = c_i)$.

- **Case (FROMSET) FROM $c_1, \ldots, c_n$ BY $\theta$:**

By Lemma 4.3.4 and Definition 4.1.1, it is easily seen that $\mathcal{M}_{after} \models e_{after}$. Now we prove that $\mathcal{M}_{after} \models \Phi_{after}$. By similar arguments in the previous case (FROMBASIC), we know that there is $\ell$ such that $1 \leq \ell \leq n$, and $\ell = \theta$, and $\llbracket c_{\text{res}} \rrbracket^\mathcal{M}_{after} = \llbracket c_\ell \rrbracket^\mathcal{M}_{after}$, and $\llbracket c_{\text{pick}} \rrbracket^\mathcal{M}_{after} = \llbracket c_\ell \rrbracket^\mathcal{M}_{after}$ for every $1 \leq j \leq m$. By Definition 4.1.1 and the rule (FROMSET), we know $\llbracket c'_{\text{res}} \rrbracket^\mathcal{M}_{after} = \llbracket c'_{\ell} \rrbracket^\mathcal{M}_{after}$ for every $m_k \leq j' \leq m$. Thus, it follows $\llbracket \text{en}(c_\ell, j', c'_{\text{pick}}) \rrbracket^\mathcal{M}_{after} = \llbracket \text{en}(c_\ell, m_k, c'_{\ell}) \rrbracket^\mathcal{M}_{after}$ for every $m_k \leq j' \leq m$. By similar arguments, we have that $\llbracket \text{en}(c_{\text{res}}, j', c'_{\text{pick}}) \rrbracket^\mathcal{M}_{after} = \llbracket \text{en}(c_{\text{res}}, m_k, c'_{\text{pick}}) \rrbracket^\mathcal{M}_{after}$ for every
4.3. Detailed proofs

By the structure of the arena $A_{after}$ and Invariant 4.2.2 for every $1 \leq j \leq m$, we have that two edges $(c_{res}, j_1, c'_{pick})$ and $(c_{i}, j_2, c'_{i})$ have the same index, that is, $j_1 = j_2$. Therefore, the constraint (INPICK) is correct under the model $M_{after}$.

- **Other cases:**

Similar to the cases (FROMBASIC) and (FROMSET).

**Lemma 4.3.16** Let $s_{before} \rightarrow s_{after}$ be a reachable transitions where the applied reduction rule is one of the rules in Section 3.10. Assume that the reduction rule replaces a sub-expression $c_{sub}$ in a formula $C_{before}$ with a fresh cell $C_{new}$. Moreover, assume that the formula $C_{before} \land \Phi_{before}$ is satisfiable. Let $M_{before}$ be a model of $C_{before} \land \Phi_{before}$, and $M_{after}$ be an extended model from $M_{before}$ as in Definition 4.1.1. Then, $M_{after}$ is a model of both $C_{after} \land \Phi_{after}$.

**Proof.** We have that Invariants 4.2.1-4.2.7 are preserved by states $s_{before}$, $s_{after}$ because $s_{before} \rightarrow s_{after}$ is reachable. Now we prove Lemma 4.3.16 by case distinction of the applied reduction rule.

- **Case** $s_{sub} = c \in C_{S}$:

The proof of this case falls naturally into two cases.

  - **Case (a)** $\llbracket c \rrbracket_{M_{before}} = \emptyset$:

    By Definition 4.1.1, we know that $\llbracket \emptyset \rrbracket_{M_{after}} = 0$, and $\llbracket c \rrbracket_{M_{after}} = \emptyset$. It follows that $\llbracket c \rrbracket_{M_{after}} \neq \llbracket c \rrbracket_{M_{after}}$ for every $1 \leq i \leq n$. It implies $\llbracket \emptyset \rrbracket_{M_{after}} \neq i$ for every $1 \leq i \leq n$. Moreover, by Invariant 4.2.2, it follows that $\llbracket c \rrbracket_{M_{after}} = \emptyset$ because $\llbracket \emptyset \rrbracket_{M_{after}} = \emptyset$. Hence, we have $M_{after} \models \Phi_{after}$.

    By the results in Chapter 2 and the standard semantics in TLA+, the value of $x$ is arbitrary, and the expression $x \in C_{S}$ is always evaluated as TRUE. By Lemma 4.3.4, we have Therefore, we have $M_{after} \models e_{after}$.

  - **Case (b)** $\llbracket c \rrbracket_{M_{before}}$ is not an empty set:

    By the above assumption and the formal semantics of the operator (ASGN), we know that $\llbracket C_{S} \rrbracket_{M_{before}} \neq \emptyset$, and $\llbracket x \rrbracket_{M_{after}} \llbracket C_{S} \rrbracket_{M_{before}}$. By Lemma 4.3.4 and Definition 4.1.1, we have $M_{after} \models e_{after}$.

Now we prove $M_{after} \models \Phi_{after}$. By Definition 4.1.1, we know that $1 \leq \llbracket \emptyset \rrbracket_{M_{after}} \leq n$. Hence, it follows that $\llbracket \emptyset \rrbracket_{M_{after}} = \emptyset$.

By Invariant 4.2.5, it follows that $\llbracket C_{S} \rrbracket_{M_{before}} \llbracket c_{i} \rrbracket_{M_{before}}$. By Definition 4.1.1, we have $\llbracket c \rrbracket_{M_{after}} = \llbracket C_{S} \rrbracket_{M_{after}}$ and $\llbracket c_{i} \rrbracket_{M_{after}} = \llbracket C_{S} \rrbracket_{M_{after}}$ for every $1 \leq i \leq n$. Hence, we have that $\llbracket C_{S} \rrbracket_{M_{after}} = \emptyset$, and $\llbracket C_{S} \rrbracket_{M_{after}} \llbracket c_{i} \rrbracket_{M_{after}}$. It implies $\llbracket \land_{1 \leq i \leq n} c_{i} \neq \llbracket C_{S} \rrbracket_{M_{after}} = \emptyset$. Therefore, we have $M_{after} \models \Phi_{after}$.
4. Soundness of the Reduction from TLA$^+$ to SMT

- **Cases** $e_{\text{sub}} = c \in \text{SUBSET } c_S$ or $e_{\text{sub}} = c \in \left[ S \rightarrow T \right]$:

  Similar to the previous case.

- **Case (SUB)** $e_{\text{sub}} = x$:

  By Lemma 4.3.4, the statement in this case is correct.

- **Case (ITE)** $e_{\text{sub}} = \text{ITE}(c_p, c_1, c_2)$:

  It is easy to see that if $\left[ \theta \right] \mathcal{M}_{\text{after}} = 1$, then $\left[ c_{\text{res}} \right] \mathcal{M}_{\text{after}} = \left[ c_1 \right] \mathcal{M}_{\text{after}}$; otherwise, $\left[ c_{\text{res}} \right] \mathcal{M}_{\text{after}} = \left[ c_2 \right] \mathcal{M}_{\text{after}}$. Therefore, by the formal semantics of the operator (ITE) and Definition 4.1.1, it immediately follows $\left[ \theta = 1 \iff c_p \right] \mathcal{M}_{\text{after}} = \text{TRUE}$.

- **Case (EXISTS)** $e_{\text{sub}} = \exists x \in S \cdot p$:

  The statement follows immediately by Lemma 4.3.4.

- **Case (CHOOSE)** $e_{\text{sub}} = \text{CHOOSE } x \in S : p$:

  By Definition 4.1.1, we have $\left[ \text{CHOOSE } x \in S : p \right] \mathcal{M}_{\text{after}} = \left[ c_{\text{res}} \right] \mathcal{M}_{\text{after}}$. By Lemma 4.3.4, we have $\mathcal{M}_{\text{after}} \models e_{\text{after}}$. Because $\text{choose}_r$ is an uninterpreted function, the introduced constraint $\text{choose}_r(c_F) = c_{\text{res}}$ does not violate the function congruency and the constraint for the operator choose in TLA$^+$ under the model $\mathcal{M}_{\text{after}}$.

Let $T$ be a set cell, and $q$ be a predicate whose free variable is also $x$. Let $T_q$ denote the set $\left\{ x \in T : q \right\}$. Assume that the set $\left[ \left\{ x \in S : p \right\} \right] \mathcal{M}_{\text{after}}$ is equal to the set $\left[ T_q \right] \mathcal{M}_{\text{after}}$. In other words, we have $\left[ \left\{ x \in S : p \right\} \right] \mathcal{M}_{\text{after}} = \left[ T_q \right] \mathcal{M}_{\text{after}}$. By the formal semantics of the operator CHOOSE in TLA$^+$, we know that $\left\{ \text{CHOOSE } x \in S : p \right\} \mathcal{M}_{\text{after}} = \left\{ \text{CHOOSE } x \in T : q \right\} \mathcal{M}_{\text{after}}$. Moreover, we have $\left[ \text{CHOOSE } x \in S : p \right] \mathcal{M}_{\text{after}} = \left[ c_{\text{res}} \right] \mathcal{M}_{\text{after}}$ by Definition 4.1.1. By the above assumption, it is safe to add constraints such as $\text{choose}_r(T_q) = c_{\text{res}}$ later. Therefore, the function congruence and the constraint related to the operator CHOOSE are preserved by the model $\mathcal{M}_{\text{after}}$ and its extended model.

- **Case (NDT)** $e_{\text{sub}} = c_1 \oplus c_2$:

  It is easily seen that the statement is correct because of the formal semantics of the operator (NDT).

**Lemma 4.3.17** Let $s$ be any state such that Invariants 4.2.1-4.2.7 are maintained in $s$, and $\mathcal{M}$ an arbitrary model of $s$. Assume that $f[a]$ is a sub-expression $e_s$, and $\left[ a \right] \mathcal{M} \in \left[ \text{DOMAIN } f \right] \mathcal{M}$, and $c_f \downarrow c_{\text{rel}} \rightarrow c_1, \ldots, c_n$. There exists an integer $i$ such that $1 \leq i \leq n$, and $\left[ \left( a, f[a] \right) \right] \mathcal{M} = \left[ c_i \right] \mathcal{M}$.

**Proof.** By Invariant 4.2.7, it follows $\left[ c_{\text{rel}} \right] \mathcal{M} = \left[ \left\{ \left( x, f(x) \right) \in \text{DOMAIN } f \right\} \mathcal{M}$. By the lemma assumption, we know $\left[ a \right] \mathcal{M} \in \left[ \text{DOMAIN } f \right] \mathcal{M}$. Therefore, $\left[ \left( a, f[a] \right) \right] \mathcal{M} \subseteq \left[ c_{\text{rel}} \right] \mathcal{M}$. By Invariant 4.2.5, it follows $\left[ c_{\text{rel}} \right] \mathcal{M} \subseteq \left\{ \left[ c_1 \right] \mathcal{M}_{\text{after}}, \ldots, \left[ c_n \right] \mathcal{M}_{\text{after}} \right\}$. Therefore, the lemma is correct. 

\[90\]
Lemma 4.3.18  Let \( s_{\text{before}} \rightarrow s_{\text{after}} \) be a reachable transition where the applied reduction rule is one of rules in Section 3.9. Assume that the formula \( e_{\text{before}} \wedge \Phi_{\text{before}} \) is satisfiable. Let \( M_{\text{before}} \) be a model of \( e_{\text{before}} \wedge \Phi_{\text{before}} \) such that \( 1 \leq \| \theta \|_{M_{\text{before}}} \leq n \), and \( M_{\text{after}} \) be an extended model from \( M_{\text{before}} \) as in Definition 4.1.1. Then, \( M_{\text{after}} \) is a model of both \( e_{\text{after}} \wedge \Phi_{\text{after}} \).

Proof. Because \( s_{\text{before}} \rightarrow s_{\text{after}} \) is a reachable transition, Invariants 4.2.1-4.2.7 are preserved by states \( s_{\text{before}}, s_{\text{after}} \). The proof of Lemma 4.3.18 is done by case distinct of the applied reduction rule.

- **Case (FUNCTOR)** \( e_{\text{sub}} = [x \in c_S \mapsto e] \):
  This case is correct by Lemma 4.3.11.

- **Case FUNDOM** \( e_{\text{sub}} = \text{DOMAIN } f \):
  In this case, constraints \( \Phi_{\text{before}}, \Phi_{\text{after}} \) are the same, that is, \( \Phi_{\text{before}} = \Phi_{\text{after}} \). By the lemma assumption, we have \( M_{\text{before}} \) is a model of \( \Phi_{\text{before}} \). By Definition 4.1.1, we have \( M_{\text{after}} \) and \( M_{\text{before}} \) are the same. Therefore, \( M_{\text{after}} \) is obviously a model of \( \Phi_{\text{after}} \). By Invariant 4.2.7, it follows \( \| \text{DOMAIN } f \|_{M_{\text{after}}} = \| \{ t [1] : t \in c_{\text{rel}} \} \|_{M_{\text{after}}} \). Therefore, \( M_{\text{after}} \) is a model of \( e_{\text{after}} \).

- **Case (FUNEXEC)** \( e_{\text{sub}} = [f \text{ except } ![a] = r] \):
  This case is correct by Lemma 4.3.12.

- **Case (FUNAPP)** \( e_{\text{sub}} = f [\text{arg}] \):
  By Definition 4.1.1, we have that \( \| (\text{arg}, f [\text{arg}]) \|_{M_{\text{after}}} = \| \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \|_{M_{\text{after}}} \), and \( 0 \leq \| \theta \|_{M_{\text{after}}} \leq n \), and 4.1.1 we know \( \| c_{\text{pair}} \|_{M_{\text{after}}} = \| \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \|_{M_{\text{after}}} \). The proof contains two cases.

  - **Case (a):** \( \| c_{\text{arg}} \|_{M_{\text{after}}} \notin \| \text{DOMAIN } f \|_{M_{\text{arg}}} \)
    In this case, by Definition 4.1.1, we know \( \| \theta \|_{M_{\text{after}}} = 0 \). Therefore, the constraint (WHENINDOMAIN) is obviously true since \( \| \theta \|_{M_{\text{after}}} = i \) is false for every \( 1 \leq i \leq n \). Now, we consider the constraint (WHENOUTSIDEDOMAIN). Let \( j \) be an arbitrary integer such that \( 1 \leq j \leq n \). If \( \| c_j \|_{M_{\text{after}}} \in \| c_{\text{rel}} \|_{M_{\text{after}}} \), it follows \( \| c_j [1] \|_{M_{\text{after}}} \in [\text{DOMAIN } c_j]_{M_{\text{after}}} \). Therefore, the constraint \( \| c_j [1] \|_{M_{\text{after}}} \notin \| c_{\text{arg}} \|_{M_{\text{after}}} \) is true under \( M_{\text{after}} \). We now assume \( \| c_j \|_{M_{\text{after}}} \notin \| c_{\text{rel}} \|_{M_{\text{after}}} \). By Invariant 4.2.7 and Lemma 4.3.14 it follows \( \| \text{en}(c_{\text{rel}}, j, c_j) \|_{M_{\text{after}}} = \text{FALSE} \). Therefore, in this case, constraint (WHENOUTSIDEDOMAIN) is also correct. Combining two cases, the result is that \( M_{\text{after}} \) is a model of \( \Phi_{\text{after}} \).

    We now consider the expression \( e_{\text{after}} \). By the definition of \( M_{\text{after}} \), it follows \( \| c_j [c_{\text{arg}}] \|_{M_{\text{after}}} = \| c_{\text{pair}} \|_{M_{\text{after}}} \). By the definition of rule (FUNAPP), we know \( e_{\text{after}} = e_{\text{before}} [c_{\text{pair}} [2] / c_j [c_{\text{arg}}]] \). By Lemma 4.3.4 it follows \( \| e_{\text{after}} \|_{M_{\text{after}}} = \| e_{\text{before}} [c_{\text{pair}} [2] / c_j [c_{\text{arg}}]] \|_{M_{\text{after}}} \). By the lemma assumption, we know \( M_{\text{before}} \) is a model of \( e_{\text{before}} \). By the definition of \( M_{\text{after}} \), we have \( M_{\text{after}} \) is also a model of \( e_{\text{before}} \). Therefore, \( M_{\text{after}} \) is a model of \( e_{\text{after}} \).
4. Soundness of the Reduction from TLA⁺ to SMT

- Case (b): \([c_{\text{arg}}]^M_{\text{after}} \in \{\text{DOMAIN } f\}^M_{\text{arg}2}\)

We define a set \(T\) as \(T \triangleq \{ \langle x, f(x) \rangle : x \in \text{DOMAIN } f \}\). Under the lemma assumption, we know \([\langle c_{\text{arg}}, f[c_{\text{arg}}] \rangle]^M_{\text{after}} \in [T]^M_{\text{after}}\). By Invariant 4.2.7 it follows \([T]^M_{\text{after}} = [c_f]^M_{\text{after}}\). Therefore, the following constraint holds \([\langle c_{\text{arg}}, f[c_{\text{arg}}] \rangle]^M_{\text{after}} \in [c_f]^M_{\text{after}}\).

By the lemma assumption and Invariant 4.2.5 it follows that

\[\mathcal{M}_{\text{after}} = \{c_1, \ldots, c_n\}\]

Therefore, by Lemma 4.3.17 there exists an integer \(j\) such that \(1 \leq j \leq n\), and \([\langle c_{\text{arg}}, f[c_{\text{arg}}] \rangle]^M_{\text{after}} = [c_j]^M_{\text{after}}\), and \([c_j]^M_{\text{after}} \in [c_f]^M_{\text{after}}\). It make the constraint \([c_j]^M_{\text{after}} = [c_{\text{arg}}]^M_{\text{after}}\) obviously true. According to Definition 4.1.1, we can assume that \([\theta]^M_{\text{after}} = j\). For every \(i \neq j\), the constraint \((\text{WHENINDOMAIN})\) is obviously true.

We now consider the case \(i = \theta = j\). By Lemma 4.3.14 it follows \([\langle c_f, i, c_i \rangle]^M_{\text{after}} \equiv ([c_i]^M_{\text{after}} \in [c_f]^M_{\text{after}}) \equiv \text{TRUE}\). Therefore, we can conclude that the constraint \((\text{WHENINDOMAIN})\) is correct for every \(0 \leq i \leq n\).

By similar arguments in the previous case, we can conclude that \(\mathcal{M}_{\text{after}}\) is a model of \(e_{\text{after}}\).

- Other cases for sequence operators:

Similar to the proofs for function operators.

\[\square\]

Lemma 4.3.19 Let \(s_{\text{before}} \rightarrow s_{\text{after}}\) be a reachable transition where the applied reduction rule is one of rules in Section 3.8. Assume that the formula \(e_{\text{before}} \land \Phi_{\text{before}}\) is satisfiable. Let \(\mathcal{M}_{\text{before}}\) be a model of \(e_{\text{before}} \land \Phi_{\text{before}}\) such that \(1 \leq [\theta]^M_{\text{before}} \leq n\), and \(\mathcal{M}_{\text{after}}\) be an extended model from \(\mathcal{M}_{\text{before}}\) as in Definition 4.1.1. Then, \(\mathcal{M}_{\text{after}}\) is a model of both \(e_{\text{after}} \land \Phi_{\text{after}}\).

\[\text{Proof.}\] The proof of Lemma 4.3.19 is similar to the proof of Lemma 4.3.18. \[\square\]

Lemma 4.3.20 Let \(s_{\text{before}} \rightarrow s_{\text{after}}\) be a reachable transition. Assume that the formula \(e_{\text{after}} \land \Phi_{\text{after}}\) is satisfiable. Let \(\mathcal{M}_{\text{after}}\) be a model of both \(e_{\text{after}} \land \Phi_{\text{after}}\). Then, \(\mathcal{M}_{\text{after}}\) is also a model of \(e_{\text{before}} \land \Phi_{\text{before}}\).

\[\text{Proof.}\] The proof of Lemma 4.3.20 is based on Invariant 4.2.4 and the semantics of the rewritten expression. \[\square\]

Theorem 4.2.2. Let \(s_0 \rightarrow \ldots \rightarrow s_m\) be a sequence of states produced by an abstract reduction system, and \(s_i = \langle e_i \mid A_i \mid \nu_i \mid \Phi_i \rangle\) for \(1 \leq i \leq m\). Assume that \(e_0\) is a formula, that is, it has type \text{Bool}. The formula \(e_0\) is satisfiable if and only if the constraint \(e_m \land \Phi_m\) is satisfiable.

\[\text{Proof.}\] We prove the direction \((\Rightarrow)\) of Theorem 4.2.2 by induction on transition steps. The basis is correct because the theorem assumption and the fact that \(\Phi_0\) is empty.
The induction step is done by case distinction of the applied reduction rule. Recall that Lemmas 4.3.14-4.3.19 are for a reachable transition, and they consider all rules in our reduction step. Therefore, the induction step immediately follows by Lemmas 4.3.14-4.3.19.

We prove the direction (⇐) of Theorem 4.2.2 by induction on transition steps and Lemma 4.3.20.

Now we can conclude that Theorem 4.2.2 is correct.

4.3.6 Finite structures

The proof of Proposition 4.1.1 requires Lemmas 4.3.21 and 4.3.22.

Lemma 4.3.21 Let e be a KERA+ expression of type \(\text{Set}[\tau]\) for some type \(\tau\), and \(\mathcal{M} = (D, I)\) be a model. If \(I\) interprets all constants and free variables in e as finite structures, then \(I\) also maps e to a finite structure.

Proof. We prove this lemma by induction on the structure of e.

- Case (\text{VAR}) e = x where x is a set variable:
  The induction step in this case is correct by the lemma assumption.

- Case (\text{CONST}) e = c where c is a constant of type \(\text{Set}[\tau]\):
  The induction step in this case is correct by the lemma assumption.

- Case (\text{DOTDOT}) e = i_1..i_2 where i_1, i_2 are integers:
  If \([i_1]^{\mathcal{M}} \leq [i_2]^{\mathcal{M}}\) then \([e]^{\mathcal{M}}\) has \((i_2 - i_1 + 1)\) elements. If not, \([e]^{\mathcal{M}}\) is the empty set. In both cases, \([e]^{\mathcal{M}}\) is a finite structure.

- Case \text{ENUM1} e = \{e_1, \ldots, e_n\} where every \(e_i\) is not a set:
  We know \([e]^{\mathcal{M}}\) has at most \(n\) elements, and none of its members are sets. Therefore, \([e]^{\mathcal{M}}\) is a finite structure.

- Case \text{ENUM2} \{e_1^{\text{set}}, \ldots, e_n^{\text{set}}\} where every \(e_i\) is a set:
  By the induction hypothesis, we know \([e_i^{\text{set}}]^{\mathcal{M}}\) is a finite structure for every i. Moreover, we have \([e]^{\mathcal{M}}\) has at most \(n\) elements which are \([e_1^{\text{set}}]^{\mathcal{M}}, \ldots, [e_n^{\text{set}}]^{\mathcal{M}}\). Therefore, \([e]^{\mathcal{M}}\) is a finite structure.

- Case (\text{UNION}) e = \text{UNION} S:
  By the induction hypothesis, we know that \(\mathcal{I}\) maps \(S\) to a finite structure. Assume that \([S]^{\mathcal{M}} = \{[S_1]^{\mathcal{M}}, \ldots, [S_k]^{\mathcal{M}}\}\) for some \(k\). Notice that \([S_i]^{\mathcal{M}}\) is also a finite structure because of the induction hypothesis for every \(1 \leq i \leq k\). Now assume that every \([S_i]^{\mathcal{M}}\) has \(n_i\) elements \(o_1^i, \ldots, o_{n_i}^i\). Again, by the induction hypothesis, we know that every \(o_j^i\) is also a finite structure. Therefore, \([\text{UNION} S_i^{\mathcal{M}}]^{\mathcal{M}}\) is a finite set with at most \(\sum_{i=1}^k n_i\)
4. Soundness of the Reduction from TLA$^+$ to SMT

elements. Moreover, every member $elem$ of $[[\text{union } S]]^M$ must be a member of some $[[S]]^M$. Because $[[S]]^M$ is a finite structure, $elem$ is so. Therefore, the induction step in this case is correct.

- Case (filer) $e = \{x \in S : p(x)\}$:

We know $[[e]]^M \subseteq [[S]]^M$. By the induction hypothesis, we know $[[S]]^M$ is a finite structure. Therefore, the induction step is correct.

- Case (map) $e = \{\text{exp}(x) : x \in S\}$:

By the induction hypothesis, we know $[[S]]^M$ is finite set. Assume that $[[S]]^M = \{o_1, \ldots, o_k\}$ where $o_i$ is an object in the domain. Therefore, $[[e]]^M$ is a finite set with at most $k$ elements, i.e. $\{[[\text{exp}]]^M(o_1), \ldots, [[\text{exp}]]^M(o_k)\}$. Let $I_1$ be an extension of $I$ with a mapping from $x$ to $o_i$. By the induction hypothesis, if $[[\text{exp}(x)]]^M$ is mapped to a set, it must be a finite structure. It implies that every member of $[[e]]^M$ is a finite structure. Moreover, we have that $[[e]]^M$ is a finite set. Therefore, the induction step is correct.

- Case (subset) $e = \text{SUBSET } S$:

In KERA$^+$, the operator $\text{subset}$ must be used together with either the set membership $\in$ or the assignment operator $\cdot$. However, the result of Lemma 4.3.21 is not restricted to the use of the operator $\text{subset}$. Here we prove that if $S$ is interpreted as a finite structure, then $\text{subset } S$ is also.

By the induction hypothesis, we know $[[S]]^M$ is a finite structure. Assume that $[[S]]^M$ has $k$ elements $o_1, \ldots, o_k$. Again, by the induction hypothesis, we know $S$’ members are also finite structures. Let $[[T]]^M$ be a subset of $[[S]]^M$. We have that $[[T]]^M$ is a finite structure every members of $[[T]]^M$ is one of finite structures $o_1, \ldots, o_k$. By the definition of the operator $\text{subset}$, we know that every member $elem$ of $[[e]]^M$ is a subset of $[[S]]^M$. Therefore, we have $elem$ is a finite structure. Moreover, we know $[[e]]^M$ is a finite set with $2^k$ elements. It implies $[[e]]^M$ is a finite structure.

- Case (funset) $e = [S \rightarrow T]$:

Similar to the operator $\text{subset}$, the operator (funset) must be used together with either the set membership $\in$ or the assignment operator $\cdot$ in KERA$^+$. However, the result of Lemma 4.3.21 is not restricted to the use of the operator (funset). Here we prove that if $S, T$ are interpreted as a finite structures, then $[S \rightarrow T]$ is also.

By the induction hypothesis, we know $[[S]]^M, [[T]]^M$ are finite structures. Assume $[[S]]^M, [[T]]^M$ have $n, m$ elements, respectively. Then $[[e]]^M$ is a finite set with $n^m$ elements. Because no elements of $[[e]]^M$ are sets, $[[e]]^M$ is a finite structure.

\textbf{Lemma 4.3.22} Let $e$ be a well-typed KERA$^+$ expression, and $M = \langle D, I \rangle$ be a model. Assume that $I$ interprets all constants and free variables in $e$ as finite structures. Then, the following hold:
4.3. Detailed proofs

1. The domain of every function in \( e \) is a finite structure. The same for every record and every tuple.

2. The co-domain of every function in \( e \) is a finite structure. The same for every record and every tuple.

3. If \( f[a] \) is a function application of type \( \text{Set}[^\tau] \) in \( e \), and \( [a]^M \) is in the domain of \( [f]^M \), then \( \mathcal{I} \) also maps \( f[a] \) to a finite structure. The same for the record application and the tuple application.

Proof. For Points (1)-(3), we show here the proof of the case function. Similar arguments applied other cases: records, and tuples. Let \( f \) be a function appearing in \( e \). While \( f \) can be defined with the operators (\text{FUNCTOR}) or (\text{FUNEXC}), we can assume that \( f = [x \in S \mapsto \text{exp}] \). The proof of Lemma 4.3.22 in this case falls naturally into three following parts.

1. By Lemma 4.3.21, it follows that \( S \) is a finite structure. Moreover, by the formal semantics of the operator \text{DOMAIN} in TLA\(^+\), we have \([\text{DOMAIN } f]^M = [S]^M\), and thus, \text{DOMAIN } f \) is also mapped to a finite structure.

2. Again, by Lemma 4.3.21, it follows that \( S \) is a finite structure. Assume that \([S]^M = \{e_1, \ldots, e_n\}\). Moreover, if \( \text{exp} \) has type \( \text{Set}[^\tau] \) for some type \( \tau \), then \([\text{exp}(e_i)]^M\) is also mapped to a finite set for every \( 1 \leq i \leq n \). Therefore, it implies that the co-domain of the function \( f \), which is the set \( \{\text{exp}(e_1), \ldots, \text{exp}(e_n)\} \) is mapped to a finite structure.

3. Assume that \([S]^M = \{e_1, \ldots, e_n\}\). Since \([a]^M \in [\text{DOMAIN } f]^M\), we know that \([f[a]]^M \in \{[\text{exp}(e_1)]^M, \ldots, [\text{exp}(e_n)]^M\}\). It follows that \([f[a]]^M \) is a finite structure by Point (2).

Therefore, Lemma 4.3.22 is correct.

Proposition 4.1.1. Let \( e \) be a KERA\(^+\) expression, and \( \mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle \) be a model. If \( \mathcal{I} \) interprets all constants and free variables in \( e \) as finite structures, then the interpretation of \( e \) is a finite structure.

Proof. By Lemma 4.3.21, the following holds: Every set expression created by a set construct in KERA\(^+\) is a finite structure. In addition to set constructs in KERA\(^+\), a set expression in \( e \) can appear as results of function applications, record applications, and tuple applications. These cases are proved in Lemma 4.3.22. Therefore, every expression in \( e \), including itself, is interpreted as a finite structure.
4.4 Summary

We have presented the detailed proofs of termination and soundness of our reduction system from TLA$^+$ to SMT. To do that, we first defined models of expressions in KerA$^+$, together with extended models. Then, we presented inductive invariants that our reduction system maintains. These inductive invariants play an important role in our proofs.

In addition, we have presented the definitions of many TLA$^+$ operators in KerA$^+$ and the additional reduction rules from TLA$^+$ to SMT that are not discussed in Chapter 3.
CHAPTER 5

Model Checking of the Skeen Atomic Multicast Algorithm

New features have been added to our model checker APALACHE since its first version described Chapter 3. In this chapter, we present our case study on model checking of Skeen's protocol with the latest versions of APALACHE and TLC by September 2021. Unlike benchmarks in Chapter 3, Skeen is an atomic multicast protocol. Atomic multicast allows messages to be sent to multiple groups of processes, while ensuring that (1) all correct addressees of every message agree either to deliver or not this message, and (2) these messages are delivered APALACHE according to some (global) total order. To verify the Skeen protocol, we first specify it in TLA++. Second, we check its properties: Integrity, Validity, Total Order, and Termination. We report on the results of running TLC and APALACHE for several combinations of fixed parameters.

This chapter adapts the short paper at PSSV’21 [TKW21].

Acknowledgments

We are grateful to Alexey Gotsman for insightful discussions on Skeen's protocol and multicast algorithms.

5.1 Introduction

Atomic multicast is a primitive that allows messages to be sent to multiple groups of processes, while ensuring that (1) all correct addressees of every message agree either to deliver or not this message, and (2) these messages are delivered according to some (global) total order. In a failure-free environment, Skeen's protocol [BJ87, Ske85] is a well-known solution to implement atomic multicast.
5. Model Checking of the Skeen Atomic Multicast Algorithm

Algorithm 5.1 Every process $p$ in Skeen’s protocol follows the below code [GLC19]

1: $\text{clock}[p] \leftarrow 0$ \hspace{1cm} \triangleright \text{Initialization}
2: \textbf{for all } $m \in \text{Msg} \text{ do}$
3: \hspace{0.5cm} $\text{delivered}[p, m] \leftarrow \text{FALSE}$
4: \hspace{0.5cm} $\text{phase}[p, m] \leftarrow \text{START}$ \hspace{0.5cm} \triangleright \text{Has not received any messages related to } m
5: \hspace{0.5cm} $\text{lts}[p, m] \leftarrow \text{NULL}$ \hspace{0.5cm} \triangleright \text{A local timestamp of } p \text{ for } m
6: \hspace{0.5cm} $\text{gts}[p, m] \leftarrow \text{NULL}$ \hspace{0.5cm} \triangleright \text{A global timestamp of } p \text{ for } m
7: \hspace{0.5cm} \textbf{when received } \text{MULTICAST}(m)$ \hspace{0.5cm} \triangleright \text{Has received a multicast message for } m
8: \hspace{1cm} $\text{clock}'[p] \leftarrow \text{clock}[p] + 1$
9: \hspace{1cm} $\text{lts}'[p, m] \leftarrow (\text{clock}, p)$ \hspace{0.5cm} \triangleright \text{Issues a local timestamp for } m
10: \hspace{1cm} $\text{phase}'[p, m] \leftarrow \text{PROPOSED}$
11: \hspace{1cm} \textbf{send } \text{PROPOSE}(p, m, \text{lts}'[p, m]) \text{ to all processes in } \text{dest}[m]$
12: \hspace{0.5cm} \textbf{when received } \text{PROPOSE}(p, m, \text{lts}'[p, m]) \text{ for every } p \in \text{dest}[m]$
13: \hspace{1cm} $\text{gts}'[p, m] \leftarrow \text{MAX}\{\text{lts}[q, m] \mid q \in \text{dest}[m]\}$ \hspace{0.5cm} \triangleright \text{Sets the global timestamp}
14: \hspace{1cm} $\text{phase}'[p, m] \leftarrow \text{COMMITTED}$
15: \hspace{1cm} $\text{clock}'[p] \leftarrow \text{MAX}\{\text{clock}[p], \text{time}(\text{gts}'[p, m])\}$ \hspace{0.5cm} \triangleright \text{Synchronizes its clock}
16: \hspace{1cm} \textbf{for all } $\{m_1 \in \text{Msg} \mid \text{phase}'[p, m_1] = \text{COMMITTED} \text{ and } \neg \text{delivered}[p, m_1]$\hspace{0.5cm}$\land \forall m_2. \text{phase}'[p, m_2] = \text{PROPOSED} \Rightarrow \text{lts}[p, m_2] > \text{gts}'[p, m_1]\}$ \text{ do}$
17: \hspace{1.5cm} $\text{delivered}'[p, m_1] \leftarrow \text{TRUE}$

In this chapter, we present our results on model checking of the Skeen protocol. Verification of the Skeen protocol faces three challenges: (i) a message can be multicast to a subset of processes, and (ii) messages must be delivered in some total order, and (iii) communication channels are FIFO queues.

To this end, we specify this protocol in TLA+. Our encoding approach tunes our model to the strength of the model checkers of TLA+: TLC [YML99] and APALACHE [KKT19] (discussed in Chapter 3). Then, we model check the properties Integrity, Validity, Total Order, and Termination with these model checkers.

Structure. We first describe Skeen’s protocol in Section 5.2. Next, we discuss its specification in TLA+ in Section 5.3. Finally, we describe the experiments in Section 5.4

5.2 The Skeen protocol

Intuitively, Skeen's protocol creates a total order on messages by assigning them unique global timestamps. A timestamp is a pair $(t, p)$ of a natural number $t > 0$, and a process identifier $p$. Given an arbitrary total order on process identifiers, timestamps are ordered lexicographically. Consequently, a total order of global timestamps can be used as a total order of messages. We let NULL denote the minimal timestamp. For a timestamp $ts = (t, p)$, we let time($ts$) = $t$. 
5.3 Specification of Skeen’s protocol in TLA$^+$

Figure 5.1 presents the pseudocode of Skeen’s protocol, which is based on the one in [GLC19]. A system instance has $N$ processes that follow Algorithm 5.1 to reach an agreement on delivered messages and their order. Processes communicate by exchanging messages through point-to-point FIFO channels. Lines 1–6 are the initialization. Every process maintains a local natural-numbered clock, which is used to generate local timestamps. Process $p$ can multicast message $m$ by sending MULTICAST($m$) to the addressees of message $m$ that is denoted by DEST[$m$]. Whenever process $p$ receives MULTICAST($m$), it increases its local clock by 1 (Line 8), issues a local timestamp for $m$ as the pair of the resulting clock value and its identifier, and stores this timestamp in lts[$p$, $m$] (Line 9). Primed variable clock$^p$ refers to the value of variable clock[$p$] in the next state. Then, process $p$ switches the status of $m$ to PROPOSED (Line 10), and sends its local timestamp in a PROPOSE message to DEST[$m$] (Line 11). After collecting the local timestamps from all addressees of message $m$, process $p$ sets the global timestamp of $m$ to the maximum of local timestamps of $m$, and stores this global timestamp in gts[$p$, $m$] (Line 13), and switches the status of $m$ to COMMITTED (Line 14). Process $p$ also synchronizes its local clock with gts[$p$, $m$] (Line 15). After that, process $p$ tries to deliver one or many committed messages. A committed message can be delivered if its global timestamp is less than the local timestamp of every proposed message (Line 18).

To be a correct implementation of atomic multicast in a failure-free environment, Skeen’s protocol needs to preserve the following properties in every execution trace [GLC19].

- **Validity.** If process $p$ delivers message $m$, then some process has multicast $m$ before, and $p \in$ DEST($m$).
- **Integrity.** Every process delivers a message at most once.
- **Total Order.** There exists a total order $\prec$ on all messages that are multicast in an execution trace such that, if process $p$ delivers message $m$, then for all messages $m' \prec m$ such that $p \in$ DEST($m'$), $p$ delivers $m'$ before $m$.
- **Termination.** If message $m$ is multicast by a process, then every process in DEST($m$) eventually delivers $m$.

5.3 Specification of Skeen’s protocol in TLA$^+$

To verify the Skeen protocol, we specify it in TLA$^+$. Our TLA$^+$ specification of the Skeen protocol is provided at the end of this chapter, pages 100–105. Encoding of the properties mentioned in Section 5.2 and related inductive invariants in TLA$^+$ can be found at [github.com/banhday/skeen]. To tune our models to the strength of the model checkers of TLA$^+$, we encode the communication channels with sets, and tag every in-transit message with a local timestamp issued by the sender. In every channel, the message with the smallest timestamp has the highest priority to be delivered to the receiver. Moreover, every received message is stored and grouped based on the identifier of the corresponding multicast message.
5. Model Checking of the Skeen Atomic Multicast Algorithm

Table 5.1: The experiments on checking Integrity, Validity, Total Order, and Termination with TLC and APALACHE. In this case, these properties are satisfied.

<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>Msgs</th>
<th>TLC</th>
<th>APALACHE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>#state</td>
<td>depth</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>( p_1 \xrightarrow{m_1} { p_1 } )</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>( p_1 \xrightarrow{m_1} { p_1, p_2 }, p_2 \xrightarrow{m_2} { p_1, p_2 } )</td>
<td>954</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>( p_1 \xrightarrow{m_1} { p_1, p_2, p_3 }, p_2 \xrightarrow{m_2} { p_2, p_3 }, p_3 \xrightarrow{m_3} { p_3, p_1 } )</td>
<td>106K</td>
<td>22</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( p_1 \xrightarrow{m_1} { p_1, p_2, p_3 }, p_2 \xrightarrow{m_2} { p_2, p_3 }, p_3 \xrightarrow{m_3} { p_3, p_1 } )</td>
<td>288K</td>
<td>26</td>
</tr>
</tbody>
</table>

Table 5.2: The experiments on checking inductive invariants IInv, VInv, and TOInv.

<table>
<thead>
<tr>
<th>N</th>
<th>C</th>
<th>Msgs</th>
<th>TLC</th>
<th>APALACHE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>tIInv</td>
<td>tVInv</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>( p_1 \xrightarrow{m_1} { p_1 } )</td>
<td>9m</td>
<td>1h36m</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>( p_1 \xrightarrow{m_1} { p_1, p_2 }, p_2 \xrightarrow{m_2} { p_1, p_2 } )</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>( p_1 \xrightarrow{m_1} { p_1, p_2, p_3 }, p_2 \xrightarrow{m_2} { p_2, p_3 }, p_3 \xrightarrow{m_3} { p_3, p_1 } )</td>
<td>TO</td>
<td>TO</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td>( p_1 \xrightarrow{m_1} { p_1, p_2, p_3 }, p_2 \xrightarrow{m_2} { p_2, p_3 }, p_3 \xrightarrow{m_3} { p_3, p_1 } )</td>
<td>TO</td>
<td>TO</td>
</tr>
</tbody>
</table>

5.4 Experiments

We first present the results of checking the properties mentioned in Section 5.2 on our TLA specification by running TLC [YML99] and APALACHE version 0.16.2 (described in Chapter 3) on different configurations of parameters. Then, we present the results of checking the inductive invariants called IInv, VInv and TInv that imply the properties Integrity, Validity, and Total Order, respectively. These inductive invariants are of the form \( S \land F \) where \( S \) is a corresponding safety predicate and \( F \) is an inductive strengthening of \( S \). Checking these inductive invariants is another way of verifying related safety properties. We run the following experiments on a machine with Core i7-6600U CPU and 16GB DDR4.

Table 5.1 summarizes the results in checking the mentioned safety and liveness properties. All properties are satisfied in this case. The column “C” shows the maximum value of local clocks. The notation \( p_1 \xrightarrow{m_1} \{ p_1, p_2 \} \) refers to that process \( p_1 \) multicasts message \( m_1 \) to a group of processes \( \{ p_1, p_2 \} \). Messages in the column “Msgs” are input parameters to the protocol. While every message is sent to all processes in the cases in Rows 2–3, no message is sent to all processes in the last case in Row 4. The column “#states” shows the number of distinct states explored by TLC. The columns “#depth” show the maximum execution length reached by TLC and APALACHE. The column “t\( \psi \)” shows the running time to check the property \( \psi \). The abbreviations “I, V, TO, Ter” mean the properties Integrity, Validity, Total Order, and Termination, respectively. The abbreviation “TO”
means a timeout of 2 hours. The notation “-” refers to that APALACHE does not currently support liveness checking.

Table 5.2 summarizes the results on checking the inductive invariants IIInv, VInv, and TOInv. The abbreviation “OOM” means out of memory. The meaning of other columns and abbreviations is the same as in Table 5.1.

5.5 Summary

As one sees from Table 5.1, TLC is more efficient than APALACHE in checking safety properties. However, Table 5.2 shows that APALACHE is faster than TLC in verifying inductive invariants. For example, TLC reaches the timeout in case of 2 processes and 2 messages (Row 3, Table 5.2), but APALACHE finishes this case in minutes.

A shortcoming of both APALACHE and TLC is that every parameter in a given specification must be fixed. Therefore, we need additional techniques to verify a specification with arbitrary parameters.

In Part II, we focus on verification techniques for the partial synchrony model [DLSSS, CT96]. A partially synchronous algorithm is parameterized in the number of processes, and parametric in time bounds for message delay and process relative speeds. We demonstrate our research ideas by using the Chandra and Toueg failure detector [CT96] as a case study since its correctness depends only on the existence of time bounds after reaching the global stabilization at some time point.

To verify the failure detector, we first define and formalize the class of symmetric point-to-point algorithms in Chapter 6. The failure detector is an instance of this class. We prove that symmetric point-to-point algorithms enjoy the cutoff property. Hence, it is sufficient to verify a symmetric point-to-point algorithm by checking only instances with few processes.

To cope with cases of arbitrary time bounds, we develop techniques to efficiently encode the failure detector, and an abstraction of in-transit messages. Then, we prove inductive invariants of the failure detector in the SMT-based interactive theorem prover IVy [MP20]. Details are provided in Chapter 7.
5. Model Checking of the Skeen Atomic Multicast Algorithm

---

**module Skeen Protocol**

**extends** Integers, FiniteSets, Sequences, TLC

**constant**

- \( @type: \text{Int}; N, \) the number of processes indexed from 1 to \( N \)
- \( @type: \text{Int}; M, \) the number of multicast messages indexed from 1 to \( M \)
- \( @type: \text{Seq(\text{Int})}; Mcaster, \) an array whose \( i \)-th element describes the multicaster of message \( i \)
- \( @type: \text{Seq(\text{Set(\text{Int})})}; GroupDest, \) an array whose \( i \)-th element describes the group of addressees of message \( i \)
- \( @type: \text{Int}; MaxClock \) the bound of local clocks

**variables**

- \( @type: \text{Int \to Int}; \) clock, the local clocks stored for each process
- \( @type: \text{\{(Int, Int) \to Int\}}; \) phase, stores the status of a message at each process
- \( @type: \text{\{(Int, Int) \to \text{[t: Int, g: Int]}\}}; \) localTS, stores the local timestamp issued by each process
- \( @type: \text{\{(Int, Int) \to \text{[t: Int, g: Int]}\}}; \) globalTS, stores the global timestamp issued by each process
- \( @type: \text{\{(Int, Int) \to \text{\{(type: Int, t: Int, id: Int, source: Int)\}}\}}; \) mcastedID, a set of messages that have been multicast
- \( @type: \text{\text{\{(Int, Int) \to \text{\{(type: Int, t: Int, id: Int, source: Int)\}}\}}}; \) receivedMcast, a set of multicast messages received by each process
- \( @type: \text{\text{\{(Int, Int) \to \text{\{(type: Int, t: Int, id: Int, source: Int)\}}\}}}; \) proposeTS, stores a set of proposals for messages
- \( @type: \text{\text{\{(Int, Int) \to \text{\{(type: Int, t: Int, id: Int, source: Int)\}}\}}}; \) delivered, a set of messages that have been delivered
- \( @type: \text{\text{\{(Int, Int) \to \text{\{(type: Int, t: Int, id: Int, source: Int)\}}\}}}; \) inTransit, a set of in-transit messages from one process to another
- \( @type: \text{\text{\{(Int, Int) \to \text{\{(type: Int, t: Int, id: Int, source: Int)\}}\}}}; \) deliver, a set of messages that have been delivered

---

Die angegebene gedruckte oder Online-Version dieser Dissertation ist an der TU Wien Bibliothek verfügbar.

The approved original version of this doctoral thesis is available in print at TU Wien Bibliothek.
5.5. Summary

\[ \texttt{@type: } \langle \text{Int, Int} \rangle \rightarrow \text{Int}; \]
\[ \texttt{dCntr} \]

\[ \texttt{\textit{vars} } \triangleq \langle \text{clock, phase, localTS, globalTS, rcvdMcast, mcastedID, inTransit, delivered, proposeTS, dCntr} \rangle \]

\[ \text{Proc} \triangleq 1 \ldots N \quad \text{set of processes} \]
\[ \text{McastID} \triangleq 1 \ldots M \quad \text{set of messages} \]
\[ \text{MType} \triangleq 10 \quad \text{type of multicast messages} \]
\[ \text{PType} \triangleq 11 \quad \text{type of proposed messages} \]
\[ \text{Start} \triangleq 12 \]
\[ \text{Proposed} \triangleq 13 \]
\[ \text{Committed} \triangleq 14 \]
\[ \text{McastMsgPhase} \triangleq \{ \text{Start, Proposed, Committed} \} \]
\[ \text{McastPhase} \triangleq [\text{McastID} \rightarrow \text{McastMsgPhase}] \]

\[ \text{TimestampNull}: \text{the init value of local timestamps and global timestamps} \]
\[ \text{Type of TimestampNull is } [t \mapsto \text{Int}, g \mapsto \text{Int}] \]
\[ \text{GroupNull} \triangleq 0 \]
\[ \text{TimeNull} \triangleq 0 \]

\[ \text{type: } [t: \text{Int}, g: \text{Int}] \]
\[ \text{TimestampNull} \triangleq [t \mapsto \text{TimeNull}, g \mapsto \text{GroupNull}] \]
\[ \text{type: Set}(\text{Int}) \]
\[ \text{Time} \triangleq 1 \ldots \text{MaxClock} \]
\[ \text{type: Set}(\text{Int}) \]
\[ \text{ProcWithNull} \triangleq 0 \ldots N \]

\[ \text{The set of all possible in-transit messages} \]
\[ \text{@type: } \text{Set}([t: \text{Int}, g: \text{Int}]); \]
\[ \text{TimestampSet} \triangleq [t: \text{Time}, g: \text{Proc}] \cup \{ \text{TimestampNull} \} \]
\[ \text{@type: } \text{Set}([\text{type: } \text{Int}, t: \text{Int}, id: \text{Int}, source: \text{Int}]); \]
\[ \text{McastMsgSet} \triangleq [t: \text{Time}, id: \text{McastID}, type: \{ \text{MType} \}, source: \text{Proc}] \]
\[ \text{@type: } \text{Set}([\text{type: } \text{Int}, t: \text{Int}, id: \text{Int}, source: \text{Int}]); \]
\[ \text{ProposeMsgSet} \triangleq [t: \text{Time}, id: \text{McastID}, type: \{ \text{PType} \}, source: \text{Proc}] \]
\[ \text{@type: } \text{Set}([\text{type: } \text{Int}, t: \text{Int}, id: \text{Int}, source: \text{Int}]); \]
\[ \text{InTransitMsgSet} \triangleq \text{McastMsgSet} \cup \text{ProposeMsgSet} \]

\[ \text{@type: } \langle \text{Int, Int} \rangle \Rightarrow \text{Int}; \]
\[ \text{Max}(a, b) \triangleq \text{IF } a > b \text{ THEN } a \text{ ELSE } b \]

\[ \text{Pick the in-transit message with the smallest timestamp from } \text{sndr} \text{ to } \text{rcvr} \]
\[ \text{@type: } \langle \text{Int, Int, [type: } \text{Int, t: Int, id: Int, source: Int}] \Rightarrow \text{Bool}; \]
\[ \text{isYoungestMsg}(\text{sndr}, \text{rcvr}, \text{msg}) \triangleq \]
\[ \forall m \in \text{inTransit}[(\text{sndr, rcvr})]: \text{msg}.t \leq m.t \]

\[ \text{Compare two timestamps based on lexicographical order} \]
5. Model Checking of the Skeen Atomic Multicast Algorithm

\[\text{Less}(t_1, t_2) \triangleq \begin{cases} t_1.t < t_2.t \\ \land t_1.t = t_2.t \\ \land t_1.g < t_2.g \end{cases}\]

The initialized state \(\text{Init}\) \(\triangleq\)
\[
\land \text{clock} = [p \in \text{Proc} \mapsto 0] \\
\land \text{phase} = [(p, m) \in \text{Proc} \times \text{McastID} \mapsto \text{Start}] \\
\land \text{localTS} = [(p, m) \in \text{Proc} \times \text{McastID} \mapsto \text{TimestampNull}] \\
\land \text{delivered} = [(p, m) \in \text{Proc} \times \text{McastID} \mapsto \text{FALSE}] \\
\land \text{proposeTS} = [(p, id) \in \text{Proc} \times \text{McastID} \mapsto \{}] \\
\land \text{mcastedID} = \{} \\
\land \text{inTransit} = [(p, q) \in \text{Proc} \times \text{Proc} \mapsto \{}] \\
\land \text{dCntr} = [(p, id) \in \text{Proc} \times \text{McastID} \mapsto 0] \\
\]

Process sender multicasts the message whose identifier is mid.

\(\text{Multicast}(\text{mid}) \triangleq\)
\[
\text{LET sender} \triangleq \text{Mcaster}[\text{mid}] \\
\text{IN} \land \text{mid} \notin \text{mcastedID} \\
\land \text{clock}[\text{sender}] < \text{MaxClock} \\
\land \text{sender} \in \text{GroupDest}[\text{mid}] \\
\land \text{mcastedID} = \text{mcastedID} \cup \{\text{mid}\} \\
\land \text{LET time} \triangleq \text{clock}[\text{sender}] + 1 \\
\land \text{IN} \land \text{inTransit}' = \text{inTransit} \cup \{\text{mcastMsg}\} \\
\text{IF} p = \text{sender} \land q \in \text{GroupDest}[\text{mid}] \\
\text{THEN} \text{inTransit}[(p, q)] \cup \{\text{mcastMsg}\} \\
\text{ELSE} \text{inTransit}[(p, q)] \\
\land \text{clock}' = [\text{clock} \text{EXCEPT} ![\text{sender}] = \text{time}] \\
\land \text{UNCHANGED} \langle \text{phase}, \text{proposeTS}, \text{rcvdMcast}, \text{localTS}, \text{globalTS}, \text{delivered}, \text{dCntr} \rangle \\
\]

Receives a multicast message

\(\text{ReceiveMulticast}(\text{sender}, \text{receiver}, \text{msg}) \triangleq\)
\[
\land \text{clock}[\text{receiver}] < \text{MaxClock} \\
\land \text{msg.type} = \text{MType} \\
\text{Deliver the message with the smallest timestamp in inTransit[sender][receiver]} \\
\land \text{isYoungestMsg}(\text{sender}, \text{receiver}, \text{msg}) \\
\land \text{rcvdMcast}' = \text{rcvdMcast} \text{EXCEPT} ![\text{receiver}] = \text{rcvdMcast}[\text{receiver}] \cup \{\text{msg.id}\} \\
\land \text{UNCHANGED} \langle \text{proposeTS}, \text{globalTS}, \text{delivered}, \text{mcastedID}, \text{dCntr} \rangle \\
\]
∧ LET mid △ msg.id
\[ time △ clock[rcver] + 1 \]
\[ newLocalTS △ \lfloor t \rightarrow time, g \mapsto rcver \rfloor \]
\[ @type: [type: \text{Int}, t: \text{Int}, id: \text{Int}, source: \text{Int}]; \]
\[ newMsg △ \lfloor type \mapsto PType, id \mapsto mid, source \mapsto rcver, t \mapsto time \rfloor \]
\[ in \quad \land \quad clock' = [\text{clock} \text{EXCEPT } ![rcver] = clock[rcver] + 1] \]
\[ \land \quad localTS' = [\text{localTS} \text{EXCEPT } ![\langle rcver, mid \rangle] = newLocalTS] \]
\[ \land \quad phase' = [\text{phase} \text{EXCEPT } ![\langle rcver, mid \rangle] = \text{Proposed}] \]

Sends its proposal to every addressee of message msg.id.
∧ IF sndr ≠ rcver
\[ \text{THEN } inTransit' = [(p, q) \in \text{Proc} \times \text{Proc} \mapsto] \]
\[ \text{IF } p = rcver \land q \in \text{GroupDest}[mid] \]
\[ \quad \text{THEN } inTransit[(p, q)] \cup \{newMsg\} \]
\[ \text{ELSE IF } p = sndr \land q = rcver \]
\[ \quad \text{THEN } inTransit[(p, q)] \setminus \{msg\} \]
\[ \quad \text{ELSE } inTransit[(p, q)] \]
\[ \text{ELSE } inTransit' = [(p, q) \in \text{Proc} \times \text{Proc} \mapsto] \]
\[ \text{IF } p = rcver \land q = rcver \]
\[ \quad \text{THEN } (inTransit[(p, q)] \cup \{newMsg\}) \setminus \{msg\} \]
\[ \text{ELSE IF } p = rcver \land q \in \text{GroupDest}[mid] \]
\[ \quad \text{THEN } inTransit[(p, q)] \cup \{newMsg\} \]
\[ \text{ELSE } inTransit[(p, q)] \]

Check whether message id can be delivered to process p.
The local timestamps of all committed messages must be greater
than the global timestamp of message id.
\[ @type: (\text{Int, Int}) \Rightarrow \text{Bool}; \]
\[ \text{CanDeliver}(p, id) \triangleq \]
\[ \land \neg \text{delivered}[(p, id)] \]
\[ \land \text{phase'}[(p, id)] = \text{Committed} \]
\[ \land \forall mid \in \text{rcvdMcast}'[p]: \]
\[ \quad (\text{phase'}[(p, mid)] = \text{Proposed} \Rightarrow \text{Less}(\text{globalTS}'[(p, id)], \text{localTS}'[(p, mid)])) \]

Process rcver has received the proposals from all addressees of message id.
\[ \text{HasAllProposes}(rcver, id) \triangleq \]
\[ \forall p \in \text{GroupDest}[id]: \exists m \in \text{proposeTS}'[(rcver, id)]: m.\text{source} = p \]

Pick a proposed message with the greatest local timestamp for message id
\[ \text{PickMsgWithMaxTS}(rcver, id) \triangleq \]
\[ \text{CHOSE } m \in \text{proposeTS}'[(rcver, id)]: \]
\[ \forall m1 \in \text{proposeTS}'[(rcver, id)]: \]
\[ \quad \lor m1.t < m.t \]
\[ \quad \lor m1.t = m.t \]
\[ \quad \land m1.\text{source} \leq m.\text{source} \]
5. Model Checking of the Skeen Atomic Multicast Algorithm

Process `rcver` has received a proposed message from process `snder`:

\[ \text{ReceivePropose}(snder, rcver, msg) \Delta \]
\[ \land msg.type = PType \]
\[ \land \text{isYoungestMsg}(snder, rcver, msg) \]
\[ \land \text{inTransit}' = \left[ \text{inTransit} \ \text{EXCEPT} \ ![(snder, rcver)] \right] \]
\[ \land \text{ts} \Delta [t \mapsto msg.t, g \mapsto msg.source] \]
\[ \land \text{id} \Delta msg.id \]
\[ \land \text{proposeTS}' = \left[ \text{proposeTS} \ \text{EXCEPT} \ ![(rcver, id)] \right] \cup \{msg\} \]
\[ \land \text{HasAllProposes}(rcver, id) \]
\[ \land \text{maxTS} \Delta \left[ g \mapsto m.source, t \mapsto m.t \right] \]
\[ \land \text{clock}' = \left[ \text{clock} \ \text{EXCEPT} \ ![rcver] \right] = \text{Max}(\text{clock}[rcver], \text{maxTS}.t) \]
\[ \land \text{phase}' = \left[ \text{phase} \ \text{EXCEPT} \ ![(rcver, id)] = \text{Committed} \right] \]
\[ \land \text{delivered}' = \left[ (p, mid) \in \text{Proc} \times \text{McastID} \mapsto \right] \]
\[ \land \text{dCntr}' = \left[ (p, mid) \in \text{Proc} \times \text{McastID} \mapsto \right] \]
\[ \land \text{Next} \Delta \]

Only to avoid deadlock checking

\[ \land \forall id \in \text{McastID} : \forall p \in \text{GroupDest}[id] : \text{delivered}[(p, id)] \]
\[ \land \text{UNCHANGED vars} \]

\[ \land \exists m \in \text{McastID} : \text{Multicast}(m) \]
\[ \land \exists p, q \in \text{Proc} : \exists m \in \text{inTransit}[(p, q)] : \]
5.5. Summary

\[
(\forall p, q, m) \ (\forall ReceiveMulticast(p, q, m) \\
\lor \forall ReceivePropose(p, q, m))
\]

\[
\lor Done
\]

\[
Fairness \triangleq WF_{vars}(\forall m \in McastID : Multicast(m) \\
\lor \exists p, q \in Proc : \exists m \in inTransit[(p, q)] : \\
\forall ReceiveMulticast(p, q, m) \\
\lor \forall ReceivePropose(p, q, m))
\]

\[
Spec \triangleq Init \land \Box[Next]_{vars} \land Fairness
\]
Part II

Symbolic Verification of Symmetric Point-to-point Distributed Algorithms
Cutoffs for Symmetric Point-to-point Distributed Algorithms

Distributed algorithms are typically parameterized in the number of participants. While in general, parameterized verification is undecidable, many distributed algorithms such as mutual exclusion, cache coherence, and distributed consensus enjoy the cutoff property, which reduces the parameterized verification problem to verification of a finite number of instances. Failure detection algorithms do not fall into one of the known classes. While consensus algorithms, for instance, are quorum-based, failure detectors typically rely on point-to-point communication and timeouts. In this chapter, we formalize this communication structure and introduce the class of symmetric point-to-point algorithms. We show that the symmetric point-to-point algorithms have a cutoff. Importantly, the cutoff properties hold in three models of computation: synchrony, asynchrony, and partial synchrony. As a result, one can verify them by model checking small instances. We demonstrate the feasibility of our approach by specifying the failure detector by Chandra and Toueg in TLA+, and by model checking them with the TLC and the APALACHE model checkers.

This chapter presents an extended version of the paper at NETYS’20 [TKW20] and partially of the paper at FORTE’21 [TKW21a] and partially of the journal paper at LMCS’23 [TKW23].

6.1 Overview

The parameterized verification problem is typically undecidable, even if every participant follows the same code [AK86, Suz88, BJK+15]. This negative result has led naturally to
two approaches of algorithm analyses: (a) semi-automated methods based on user-guided invariants and proof assistants, and (b) automatic techniques for restricted classes of algorithms and properties. A particularly fascinating case is the cutoff property that guarantees that analyzing a few small instances is necessary and sufficient to reason about the correctness of all instances \cite{EN95,CTTV04}. In a nutshell, given a property $\xi$ and a system that has a parameter $m$, the system has a cutoff $B \geq 1$, if whenever all instances that assign a value not greater than $B$ to a parameter $m$ satisfy $\xi$, then all instances which assign an arbitrary number to $m$ satisfy $\xi$. Hence, verification of algorithms that enjoy the cutoff property can be done by model checking of finite instances.  

Our motivation comes from the Chandra and Toueg failure detector, which we discussed in Section 1.2. In this chapter, we introduce the class of symmetric point-to-point algorithms that contains the failure detector. Importantly, we show that this class enjoys the cutoff property. Informally, an instance in this class contains $N$ processes that follow the same algorithm, and communicate with each other by sending and receiving messages through point-to-point communication channels. At each process, local memory can be partitioned into regions such that one region corresponds one-to-one with a remote process, e.g., the array element $\text{timeout}[p, q]$ at a process $p$ stores the maximum waiting time for a process $q$ by the process $p$. The failure detector \cite{CT96} is one example of this class. Let $1..N$ be a set of indexes. We show two cutoffs for these algorithms:

1. Let $i$ be an index, and $\omega_{\{i\}}$ be an LTL\X (the stuttering-insensitive linear temporal logic) formula in which every predicate takes one of the forms: $P_1(i)$ or $P_2(i, i)$. Properties of the form $\bigwedge_{i \in 1..N} \omega_{\{i\}}$ has a cutoff of 1.

2. Let $i$ and $j$ be different indexes, and $\psi_{\{i,j\}}$ be an LTL\X formula in which every predicate takes one of the (syntactic) forms: $P_1(i)$, or $P_2(j)$, or $P_3(i, j)$, or $P_4(j, i)$. Properties of the form $\bigwedge_{i,j \in 1..N} \psi_{\{i,j\}}$ has a cutoff of 2.

For instance, by the second cutoff result, we can verify the following property called the strong completeness property of the failure detector in \cite{CT96} by model checking of an instance of size 2.

$$\mathbf{F} \mathbf{G} (\forall i, j \in 1..N : (\text{Correct}(i) \land \neg \text{Correct}(j)) \Rightarrow \text{Suspected}(i, j))$$

This formula means that every crashed process is eventually permanently suspected by every correct process. We are writing $\mathbf{F}$ and $\mathbf{G}$ to denote “eventually” and “globally” operators of linear temporal logic (LTL), see \cite{CJGK+18}. We demonstrate the feasibility of our approach by specifying Chandra and Toueg’s failure detectors \cite{CT96} in the language TLA+ \cite{Lam02}, and verifying the specification with two model checkers: TLC \cite{YML99} and APALACHE (described in Chapter 3).

**Structure.** Section 6.2 defines the model of computation as a transition system. Section 6.3 shows our main contributions: two cutoff results in the class of symmetric point-to-point distributed algorithms under asynchrony. Section 6.4 presents the detailed
6.2. Model of Computation

\textbf{Algorithm 1.1} The eventually perfect failure detector algorithm in [CT96]

\begin{algorithm}
\begin{algorithmic}[1]
\State \textit{Every process} \( p \in 1..N \) \textit{executes the following:}
\ForAll{\( q \in 1..N \)} \Comment{Initialization step}
\State \textit{timeout} \( [p, q] := \text{default-value} \)
\State \( \text{suspected} \ [p, q] := \bot \)
\EndFor
\State Send “alive” to all \( q \in 1..N \) \Comment{Task 1: repeat periodically}
\ForAll{\( q \in 1..N \)} \Comment{Task 2: repeat periodically}
\If{\textit{suspected} \( [p, q] = \bot \) \textbf{and} \textit{not hear} \( q \) during last \textit{timeout} \( [p, q] \) ticks}
\State \textit{suspected} \( [p, q] := \top \)
\EndIf
\EndFor
\If{\textit{suspected} \( [p, q] \)} \Comment{Task 3: when receive “alive” from \( q \)}
\EndIf
\State \textit{timeout} \( [p, q] := \text{timeout} \ [p, q] + 1 \)
\State \textit{suspected} \( [p, q] := \bot \)
\end{algorithmic}
\end{algorithm}

Proofs of our cutoff results under asynchrony. Section 6.5 shows our extended cutoff results under the partial synchrony model with unknown bounds \( \Delta \) and \( \Phi \). Section 6.6 presents how we encode the model of computation, and the Chandra and Toueg failure detector [CT96] in TLA\(^+\), and the model checking results.

We repeat Algorithm 1.1 in Section 1.2 to help the reader follow research ideas in the following sections.

6.2 Model of Computation

In this section, we introduce the class of symmetric point-to-point algorithms, and present how to formalize such algorithms as transition systems. Since every process follows the same algorithm, we first define a process template that captures the process behavior in Section 6.2.1. Every process is an instance of the process template.

In Section 6.2.2, we present the formalization of the global system. This formalization is adapted with the time constraints under partial synchrony in Section 6.2.3 and our analysis is for the model under partial synchrony.

Intuitively, a global system is a composition of \( N \) processes, \( N^2 \) point-to-point outgoing message buffers, and \( N \) control components that capture what processes can take a step. Every process is identified with a unique index in 1..\( N \), and follows the same deterministic algorithm. Moreover, a global system allows: (i) multiple processes to take (at most) one step in one global step, and (ii) some processes to crash. Every process may execute three kinds of transitions: \textit{internal}, \textit{round}, and stuttering. Notice that in one global step, some processes may send a message to all, and some may receive messages and do computation. Hence, we need to decide which processes move, and what happens to the message buffers. We introduce four sub-rounds: \textit{Schedule}, \textit{Send}, \textit{Receive}, and \textit{Computation}. The transitions for these sub-rounds are called internal ones. A global round transition is a composition of four internal transitions. We formalize sub-rounds
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

and global steps later. As a result of modeling, there exists an arbitrary sequence of global configurations which is not accepted in asynchrony. So, we define so-called admissible sequences of global configurations under asynchrony.

Recall that the network topology of algorithms in the symmetric point-to-point class contains $N^2$ point-to-point message buffers. Every transposition on a set of process indexes preserves the network topology. Importantly, every transposition on process indexes also preserves the structures of both the process template and the global transition system. It implies that both the process template and the global transition system are symmetric.

6.2.1 The Process Template

We fix a set of process indexes as $1..N$. Moreover, we assume that the message content does not have indexes of its receiver and sender. We let $\text{Msg}$ denote a set of potential messages, and $\text{Set}(\text{Msg})$ denote the set of sets of messages.

We model a process template as a transition system $\mathcal{U}_N = (Q_N, \mathcal{T}_N, \mathcal{R}_N, q_0^N)$ where

$$Q_N = \text{Loc} \times \text{Set(Msg)} \times \ldots \times \text{Set(Msg)} \times \mathcal{D} \times \ldots \times \mathcal{D}$$

is a set of template states\(^1\). $\mathcal{T}_N$ is a set of template transitions, $\mathcal{R}_N \subseteq Q_N \times \mathcal{T}_N \times Q_N$ is a template transition relation, and $q_0^N \in Q_N$ is an initial state. These components of $\mathcal{U}_N$ are defined as follows.

States. A template state $\rho$ is a tuple $(\ell, S_1, \ldots, S_N, d_1, \ldots, d_N)$ where:

- $\ell \in \text{Loc}$ refers to the value of a program counter that ranges over a set $\text{Loc}$ of locations. We assume that $\text{Loc} = \text{Loc}_{\text{snd}} \cup \text{Loc}_{\text{rcv}} \cup \text{Loc}_{\text{comp}} \cup \{\ell_{\text{crash}}\}$, and three sets $\text{Loc}_{\text{snd}}$, $\text{Loc}_{\text{rcv}}$, $\text{Loc}_{\text{comp}}$ are disjoint, and $\ell_{\text{crash}}$ is a special location of crashes. To access the program counter, we use a function $pc: Q_N \rightarrow \text{Loc}$ that takes a template state at its input, and produces its program counter as the output. Let $\rho(k)$ denote the $k^{\text{th}}$ component in a template state $\rho$. For every $\rho \in Q_N$, we have $pc(\rho) = \rho(1)$.

- $S_i \in \text{Set}(\text{Msg})$ refers to a set of messages. It is to store the messages received from a process $p_i$ for every $i \in 1..N$. To access a set of received messages from a particular process whose index is in $1..N$, we use a function $\text{rcvd}: Q_N \times 1..N \rightarrow \text{Set}(\text{Msg})$ that takes a template state $\rho$ and a process index $i$ at its input, and produces the $(i+1)^{\text{th}}$ component of $\rho$ at the output, i.e. for every $\rho \in Q_N$, we have $\text{rcvd}(\rho, i) = \rho(1+i)$.

- $d_i \in \mathcal{D}$ refers to a local variable related to a process $p_i$ for every $i \in 1..N$. To access a local variable related to a particular process whose index in $1..N$, we use a

\(^1\)We denote $S_1 \times \ldots \times S_m$ by a set $\{ (s_1, \ldots, s_m) \mid \bigwedge_{1 \leq i \leq m} s_i \in S_i \}$ of tuples. The elements of the set $Q_N$ are tuples with $2N + 1$ elements.
6.2. Model of Computation

function \texttt{lvar}: \mathcal{Q}_N \times \mathbb{N} \rightarrow \mathcal{D} that takes a template state \( \rho \) and a process index \( i \) at its input, and produces the \((1 + N + i)^{th}\) component of \( \rho \) as the output, i.e. \( \texttt{lvar}(\rho, i) = \rho(1 + N + i) \) for every \( \rho \in \mathcal{Q}_N \). For example, for every process \( p \) in Algorithm 1.1, variable \( d_i \) of process \( p \) refers to a tuple of the two variables \( \text{timeout}[p, i] \) and \( \text{suspect}[p, i] \).

\textbf{Initial state.} The initial state \( q^0_N \) is a tuple \( q^0_N = (\ell_0, \emptyset, \ldots, \emptyset, d_0, \ldots, d_0) \) where \( \ell_0 \) is a location, every box for received messages is empty, and every local variable is assigned a constant \( d_0 \in \mathcal{D} \).

\textbf{Transitions.} We define \( Tr_N = \text{CSnd} \cup \text{CRcv} \cup \{ \text{comp}, \text{crash}, \text{stutter} \} \) where

- \( \text{CSnd} \) is a set of transitions. Every transition in \( \text{CSnd} \) refers to a task that does some internal computation, and sends a message to all. For example, in task 1 in Algorithm 1.1, a process increases its local clock, and performs an instruction to send “alive” to all. We let \( \text{csnd}(m) \) denote a transition referring to a task with an action to send a message \( m \in \text{Msg} \) to all.

- \( \text{CRcv} \) is a set of transitions. Every transition in \( \text{CRcv} \) refers to a task that receives \( N \) sets of messages, and does some internal computation. For example, in task 2 in Algorithm 1.1 a process increases its local clock, receives messages, and removes false-negative predictions. We let \( \text{crcv}(S_1, \ldots, S_N) \) denote a transition referring to a task with an action to receive sets \( S_1, \ldots, S_N \) of messages. These sets \( S_1, \ldots, S_N \) are delivered by the global system.

- \( \text{comp} \) is a transition which refers to a task with purely local computation. In other words, this task has neither send actions nor receive actions.

- \( \text{crash} \) is a transition for crashes.

- \( \text{stutter} \) is a transition for stuttering steps.

\textbf{Transition relation.} For two states \( \rho, \rho' \in \mathcal{Q}_N \) and a transition \( tr \in Tr_N \), instead of \( (\rho, tr, \rho') \), we write \( \rho \xrightarrow{tr} \rho' \). In the model of [DLS88, CT96], each process follows the same deterministic algorithm. Hence, we assume that for every \( \rho_0 \xrightarrow{tr_0} \rho'_0 \) and \( \rho_1 \xrightarrow{tr_1} \rho'_1 \), if \( \rho_0 = \rho_1 \) and \( tr_0 = tr_1 \), then it follows that \( \rho'_0 = \rho'_1 \). Moreover, we assume that there exist the following functions which are used to define constraints on the template transition relation:

- A function \( \text{nextLoc}: \text{Loc} \rightarrow \text{Loc} \) takes a location at its input and produces the next location as the output.

- A function \( \text{genMsg}: \text{Loc} \rightarrow \text{Set}(\text{Msg}) \) takes a location as its input, and produces a singleton set that contains the message that is sent to all processes in the current
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

task. The output can be an empty set. For example, if a process is performing a Receive task, the output of \( \text{genMsg} \) is an empty set.

- A function \( \text{nextVar} : \text{Loc} \times \text{Set}(\text{Msg}) \times \mathcal{D} \rightarrow \mathcal{D} \) takes a location, a set of messages, and a local variable’s value, and produces a new value of a local variable as the output.

Let us fix functions \( \text{nextLoc}, \text{genMsg} \) and \( \text{nextVar} \). We define the template transitions as follows.

1. For every message \( m \in \text{Msg} \), for every pair of states \( \rho, \rho' \in Q_N \), we have \( \rho \xrightarrow{\text{csnd}(m)} \rho' \) if and only if
   a) \( pc(\rho) \in \text{Loc}_{\text{csnd}} \land pc(\rho') = \text{nextLoc}(pc(\rho)) \land \{ m \} = \text{genMsg}(pc(\rho)) \)
   b) \( \forall i \in 1..N : \text{rcvd}(\rho, i) = \text{rcvd}(\rho', i) \)
   c) \( \forall i \in 1..N : \text{lvar}(\rho', i) = \text{nextVar}(pc(\rho), \emptyset, \text{lvar}(\rho, i)) \)

Constraint (a) implies that the update of a program counter and the construction of a sent message \( m \) depend on only the current value of a program counter, and a process sends only \( m \) to all in this step. For example, process \( p \) in Algorithm 1.1 sends only message “alive” in Task 1. Constraint (b) refers to that no message was delivered. Constraint (c) implies that the value of \( \text{lvar}(\rho', i) \) depends only on the current location and the value of \( \text{lvar}(\rho, i) \). The empty set in Constraint (c) means that no messages have been delivered.

2. For arbitrary sets of messages \( S_1, \ldots, S_N \subseteq \text{Msg} \), for every pair of states \( \rho, \rho' \in Q_N \), we have \( \rho \xrightarrow{\text{crcv}(S_1, \ldots, S_N)} \rho' \) if and only if the following constraints hold:
   a) \( pc(\rho) \in \text{Loc}_{\text{crcv}} \land pc(\rho') = \text{nextLoc}(pc(\rho)) \land \emptyset = \text{genMsg}(pc(\rho)) \)
   b) \( \forall i \in 1..N : \text{rcvd}(\rho', i) = \text{rcvd}(\rho, i) \cup S_i \)
   c) \( \forall i \in 1..N : \text{lvar}(\rho', i) = \text{nextVar}(pc(\rho), S_i, \text{lvar}(\rho, i)) \)

Constraint (a) in \( \text{crcv} \) is similar to constraint (a) in \( \text{csnd} \), except that no message is sent in this sub-round. Constraint (b) refers that messages in a set \( S_i \) are from a process indexed \( i \), and have been delivered in this step. For example, in Algorithm 1.1 Constraint (b) implies that \( \text{rcvd}(\rho, i) \subseteq \{\text{“alive”}\} \) for every template state \( \rho \) and every index \( 1 \leq i \leq N \). After the first “alive” message was received, the value of \( \text{rcvd}(\rho, i) \) is unchanged. This does not raise any issues in our analysis as Line 7 in Algorithm 1.1 considers only how long process \( p \) has waited for a new message from process \( q \). Constraint (c) in \( \text{crcv} \) implies that the value of \( \text{lvar}(\rho', i) \) depends on only the current location, the set \( S_i \) of messages that have been delivered, and the value of \( \text{lvar}(\rho, i) \).
3. For every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{\text{comp}} \rho'$ if and only if the following constraints hold:

   a) $pc(\rho) \in \text{Loc}_{\text{comp}} \land pc(\rho') = \text{nextLoc}(pc(\rho)) \land \emptyset = \text{genMsg}(pc(\rho))$
   
   b) $\forall i \in 1..N: \text{rcvd}(\rho', i) = \text{rcvd}(\rho, i)$
   
   c) $\forall i \in 1..N: \text{lvar}(\rho', i) = \text{nextVar}(pc(\rho), \emptyset, lvar(\rho, i))$

Hence, this step has only local computation. No message is sent or delivered.

4. For every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{\text{crash}} \rho'$ if and only if the following constraints hold:

   a) $pc(\rho) \neq \ell_{\text{crash}} \land pc(\rho') = \ell_{\text{crash}}$
   
   b) $\forall i \in 1..N: \text{rcvd}(\rho', i) = \text{rcvd}(\rho, i) \land \text{lvar}(\rho', i) = \text{lvar}(\rho, i)$

Only the program counter is updated by switching to $\ell_{\text{crash}}$.

5. For every pair of states $\rho, \rho' \in Q_N$, we have $\rho \xrightarrow{\text{stutter}} \rho'$ if and only if $\rho = \rho'$.

### 6.2.2 Modeling the Global Distributed Systems

We now present the formalization of the global system. In this model, multiple processes might take a step in a global step. This characteristic allows us to extend this model with partial synchrony constraints that are formalized in Section 6.2.3. To capture the semantics of asynchrony, we simply need a constraint that only one process can take a step in a global step [AW04]. This constraint is formalized in the end of this subsection.

Given $N$ processes which are instantiated from the same process template $\mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_0^N)$, the global system is a composition of (i) these processes, and (ii) $N^2$ point-to-point buffers for in-transit messages, and (iii) $N$ control components that capture what processes can take a step. We formalize the global system as a transition system $\mathcal{G}_N = (\mathcal{C}_N, T_N, R_N, g_0^N)$ where

- $\mathcal{C}_N = (Q_N)^N \times \text{Set}(\text{Msg})^{N \times N} \times \text{Bool}^N$ is a set of global configurations,
- $T_N$ is a set of global \emph{internal}, \emph{round}, and stuttering transitions,
- $R_N \subseteq \mathcal{C}_N \times T_N \times \mathcal{C}_N$ is a global transition relation, and
- $g_0^N$ is an initial configuration.

These components are defined as follows.
Configurations. A global configuration $\kappa$ is defined as a following tuple

$$\kappa = (q_1, \ldots, q_N, S^1_1, S^2_1, \ldots, S^N_1, \ldots, S^1_N, \ldots, S^N_N, act_1, \ldots, act_N)$$

where:

- $q_i \in Q_N$: This component is a state of a process $p_i$ for every $i \in 1..N$. To access a local state of a particular process, we use a function $lstate: C_N \times 1..N \to Q_N$ that takes input as a global configuration $\kappa$ and a process index $i$, and produces output as the $i^{th}$ component of $\kappa$ which is a state of a process $p_i$. Let $\kappa(i)$ denote the $i^{th}$ component of a global configuration $\kappa$. For every $i \in 1..N$, we have $lstate(\kappa, i) = \kappa(i) = q_i$.

- $S^s_r \in \text{Set}(\text{Msg})$: This component is a set of in-transit messages from a process $p_s$ to a process $p_r$ for every $s, r \in 1..N$. To access a set of in-transit messages between two processes, we use a function $buf: C_N \times 1..N \times 1..N \to \text{Set}(\text{Msg})$ that takes input as a global configuration $\kappa$, and two process indexes $s, r$, and produces output as the $(s \cdot N + r)^{th}$ component of $\kappa$ which is a message buffer from a process $p_s$ (sender) to a process $p_r$ (receiver). Formally, we have $buf(\kappa, s, r) = \kappa((s \cdot N + r)\cdot N + i) = S^s_r$ for every $s, r \in 1..N$.

- $act_i \in \text{Bool}$: This component says whether a process $p_i$ can take one step in a global step for every $i \in 1..N$. To access a control component, we use a function $active: C_N \times 1..N \to \text{Bool}$ that takes input as a configuration $\kappa$ and a process index $i$, and produces output as the $((N+1)\cdot N + i)^{th}$ component of $\kappa$ which refers to whether a process $p_i$ can take a step. Formally, we have $active(\kappa, i) = \kappa((N+1)\cdot N + i)$ for every $i \in 1..N$. The environment sets the values of $act_1, \ldots, act_N$ in the sub-round Schedule defined later.

We will write $\kappa \in (Q_N)^N \times \text{Set}(\text{Msg})^{N-N} \times \text{Bool}^N$ or $\kappa \in C_N$.

**Initial configuration.** The global system $G_N$ has one initial configuration $g^0_N$, and it must satisfy the following constraints:

1. $\forall i \in 1..N: \neg active(g^0_N, i) \wedge lstate(g^0_N, i) = q^0_i$

2. $\forall s, r \in 1..N: buf(g^0_N, s, r) = \emptyset$

**Global stuttering transition.** We extend the relation $\sim$ with stuttering: for every configuration $\kappa$, we allow $\kappa \sim \kappa$. The stuttering transition is necessary in the proof of Lemma [6.3.5] presented in Section [6.3]
Global internal transitions. In the model of [DLSS88, CT96], many processes can take a step in a global step. We assume that a computation of the distributed system is organized in rounds, i.e., global ticks, and every round is organized as four sub-rounds called Schedule, Send, Receive, and Computation. To model that as a transition system, for every sub-round we define a corresponding transition: for the sub-round Schedule, for the sub-round Send, for the sub-round Receive, for the sub-round Comp. These transitions are called global internal transitions. We define the semantics of these sub-rounds as follows.

1. Sub-round Schedule. The environment starts with a global configuration where every process is inactive, and move to another by non-deterministically deciding what processes become crashed, and what processes take a step in the current global step. Every correct process takes a stuttering step, and every faulty process is inactive. If a process \( p \) is crashed in this sub-round, every incoming message buffer to \( p \) is set to the empty set. Formally, for \( \kappa, \kappa' \in C_N \), we have \( \kappa \xrightarrow{\text{Sched}} \kappa' \) if the following constraints hold:
   
a) \( \forall i \in 1..N: \neg \text{active}(\kappa, i) \)
   
b) \( \forall i \in 1..N: \text{lstate}(\kappa, i) \xrightarrow{\text{stutter}} \text{lstate}(\kappa', i) \lor \text{lstate}(\kappa, i) \xrightarrow{\text{crash}} \text{lstate}(\kappa', i) \)
   
c) \( \forall i \in 1..N: \text{pc}(\text{lstate}(\kappa', i)) = \ell_{\text{crash}} \Rightarrow \neg \text{active}(\kappa', i) \)
   
d) \( \forall s, r \in 1..N: \text{pc}(\text{lstate}(\kappa', r)) \neq \ell_{\text{crash}} \Rightarrow \text{buf}(\kappa, s, r) = \text{buf}(\kappa', s, r) \)
   
e) \( \forall r \in 1..N: \text{pc}(\text{lstate}(\kappa', r)) = \ell_{\text{crash}} \Rightarrow (\forall s \in 1..N: \text{buf}(\kappa', s, r) = \emptyset) \)

We let predicate \( \text{Enabled}(\kappa, i, L) \) denote whether process \( i \) whose location at the configuration \( \kappa \) is in \( L \) takes a step from \( \kappa \). Formally, we have

\[
\text{Enabled}(\kappa, i, L) \triangleq \text{active}(\kappa, i) \land \text{pc}(\text{lstate}(\kappa, i)) \in L
\]

Predicate \( \text{Enabled} \) is used in the definitions of other sub-rounds.

2. Sub-round Send. Only processes that perform send actions can take a step in this sub-round. Such processes become inactive at the end of this sub-round. Fresh sent messages are added to corresponding message buffers. To define the semantics of the sub-round Send, we use the following predicates:

\[
\text{Frozen}_S(\kappa, \kappa', i) \triangleq \text{lstate}(\kappa, i) \xrightarrow{\text{stutter}} \text{lstate}(\kappa', i)
\land \text{active}(\kappa, i) = \text{active}(\kappa', i)
\land \forall r \in 1..N: \text{buf}(\kappa, i, r) = \text{buf}(\kappa', i, r)
\]

\[
\text{Sending}(\kappa, \kappa', i, m) \triangleq \forall r \in 1..N: m \notin \text{buf}(\kappa, i, r)
\land \forall r \in 1..N: \text{buf}(\kappa', i, r) = \{m\} \cup \text{buf}(\kappa, i, r)
\land \text{lstate}(\kappa, i) \xrightarrow{\text{send}(m)} \text{lstate}(\kappa', i)
\]

Formally, for \( \kappa, \kappa' \in C_N \), we have \( \kappa \xrightarrow{\text{Send}} \kappa' \) if the following constraints hold:
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

a) \( \forall i \in 1..N : \neg Enabled(\kappa, i, Loc_{snd}) \Rightarrow Frozen_S(\kappa, \kappa', i) \)

b) \( \forall i \in 1..N : Enabled(\kappa, i, Loc_{snd}) \Leftrightarrow \exists m \in \text{Msg}: Sending(\kappa, \kappa', i, m) \)

c) \( \forall i \in 1..N : Enabled(\kappa, i, Loc_{snd}) \Rightarrow \neg active(\kappa', i) \)

The semantics of the Send sub-round forces that the Send primitive is atomic.

3. Sub-round Receive. Only processes that perform receive actions can take a step in this sub-round. Such processes become inactive at the end of this sub-round. Sets of delivered messages that may be empty are removed from corresponding message buffers. To define the semantics of this sub-round, we use the following predicates:

\[ Frozen_R(\kappa, \kappa', i) \triangleq \text{lstate}(\kappa, i) \xrightarrow{\text{stutter}} \text{lstate}(\kappa', i) \]
\[ Receiving(\kappa, \kappa', i, S_1, \ldots, S_N) \triangleq \forall s \in 1..N : \text{buf}(\kappa', s, i) = \emptyset \]

Formally, for \( \kappa, \kappa' \in C_N \), we have \( \kappa \xrightarrow{\text{rcv}} \kappa' \) if the following constraints hold:

a) \( \forall i \in 1..N : \neg Enabled(\kappa, i, Loc_{rcv}) \Rightarrow Frozen_R(\kappa, \kappa', i) \)

b) \( \forall i \in 1..N : Enabled(\kappa, i, Loc_{rcv}) \Leftrightarrow \exists S_1, \ldots, S_N \subseteq \text{Msg} : Receiving(\kappa, \kappa', i, S_1, \ldots, S_N) \)

c) \( \forall i \in 1..N : Enabled(\kappa, i, Loc_{rcv}) \Rightarrow \neg active(\kappa', i) \)

4. Sub-round Computation. Only processes that perform internal computation actions can take a step in this sub-round. Such processes become inactive at the end of this sub-round. Every message buffer is unchanged. Formally, for \( \kappa, \kappa' \in C_N \), we have \( \kappa \xrightarrow{\text{comp}} \kappa' \) if the following constraints hold:

a) \( \forall i \in 1..N : Enabled(\kappa, i, Loc_{comp}) \Leftrightarrow \text{lstate}(\kappa, i) \xrightarrow{\text{comp}} \text{lstate}(\kappa', i) \)

b) \( \forall i \in 1..N : \neg Enabled(\kappa, i, Loc_{comp}) \Leftrightarrow \text{lstate}(\kappa, i) \xrightarrow{\text{stutter}} \text{lstate}(\kappa', i) \)

c) \( \forall s, r \in 1..N : \text{buf}(\kappa', s, r) = \text{buf}(\kappa', s, r) \)

d) \( \forall i \in 1..N : Enabled(\kappa, i, Loc_{comp}) \Rightarrow \neg active(\kappa', i) \)

Remark 6.2.1. Predicate Sending refers that at most one “alive” message in Algorithm 1.1 is in every message buffer. In Section 6.2.3 we extend our formalization by introducing the notion of time and by modeling time constraints under partial synchrony. In that
formalization, every “alive” message is tagged with its age, and therefore, the message buffers can have multiple messages.

Remark 6.2.2. Observe that the definitions of $\kappa \xrightarrow{\text{Sched}} \kappa'$, and $\kappa \xrightarrow{\text{Snd}} \kappa'$, and $\kappa \xrightarrow{\text{Rec}} \kappa'$, and $\kappa \xrightarrow{\text{Comp}} \kappa'$ allow $\kappa = \kappa'$, that is stuttering. This captures, e.g., global steps in [DLS88, CT96] where no process sends a message.

Global round transitions. Intuitively, every global round transition is induced by a sequence of four transitions: a $\xrightarrow{\text{Sched}}$ transition, a $\xrightarrow{\text{Snd}}$ transition, a $\xrightarrow{\text{Rec}}$ transition, and a $\xrightarrow{\text{Comp}}$ transition. We let $\leadsto$ denote global round transitions. For every pair of global configurations $\kappa_0, \kappa_4 \in C_N$, we say $\kappa_0 \leadsto \kappa_4$ if there exist three global configurations $\kappa_1, \kappa_2, \kappa_3 \in C_N$ such that $\kappa_0 \xrightarrow{\text{Sched}} \kappa_1 \xrightarrow{\text{Snd}} \kappa_2 \xrightarrow{\text{Rec}} \kappa_3 \xrightarrow{\text{Comp}} \kappa_4$. Moreover, global round transitions allow some processes to crash only in the sub-round Schedule. We call these faults clean crashes. Notice that correct process $i$ can make at most one global internal transition in every global round transition since the component $\text{act}_i$ is false after process $i$ makes a transition.

Admissible sequences. An infinite sequence $\pi = \kappa_0 \kappa_1 \ldots$ of global configurations in $G_N$ is admissible if the following constraints hold:

1. $\kappa_0$ is the initial state, i.e. $\kappa_0 = g^0$, and
2. $\pi$ is stuttering equivalent with an infinite sequence $\pi' = \kappa'_0 \kappa'_1 \ldots$ such that $\kappa'_{4k} \xrightarrow{\text{Sched}} \kappa'_{4k+1} \xrightarrow{\text{Snd}} \kappa'_{4k+2} \xrightarrow{\text{Rec}} \kappa'_{4k+3} \xrightarrow{\text{Comp}} \kappa'_{4k+4}$ for every $k \geq 0$.

Notice that it immediately follows by this definition that if $\pi = \kappa_0 \kappa_1 \ldots$ is an admissible sequence of configurations in $G_N$, then $\kappa'_{4k} \leadsto \kappa'_{4k+4}$ for every $k \geq 0$. From now on, we only consider admissible sequences of global configurations.

Admissible sequences under synchrony. Let $\pi = \kappa_0 \kappa_1 \ldots$ be an admissible sequence of global configurations in $G_N$. As every correct process makes a transition in every global step under synchrony [AW04], we say that $\pi$ is under synchrony if every correct process is active after a sub-round Schedule. Formally, for every transition $\kappa \xrightarrow{\text{Sched}} \kappa'$ in $\pi$, the following constraint holds: $\forall i \in 1..N: \text{pc}(\text{Istate}(\kappa', i)) \neq \ell_{\text{crash}} \Rightarrow \text{active}(\kappa', i)$.

Admissible sequences under asynchrony. Let $\pi = \kappa_0 \kappa_1 \ldots$ be an admissible sequence of global configurations in $G_N$. As at most one process can make a transition in every global step under asynchrony [AW04], we say that $\pi$ is under asynchrony if at most one process is active after a sub-round Schedule. Formally, for every transition $\kappa \xrightarrow{\text{Sched}} \kappa'$ in $\pi$, the following constraint holds: $\forall i, j \in 1..N: \text{active}(\kappa', i) \land \text{active}(\kappa', j) \Rightarrow i = j$. 

6.2.3 Modeling Time Constraints under Partial Synchrony

Time parameters in partial synchrony only reduce the execution space compared to asynchrony. Hence, we can formalize the system behaviors under partial synchrony by extending the above formalization of the system behaviors with the notion of time, message ages, time constraints, and admissible sequences of configurations under partial synchrony. They are defined as follows.

**Time.** Time is progressing with global round transitions. Formally, let \( \pi = \kappa_0 \kappa_1 \ldots \) be an admissible sequence of global configurations in \( \mathcal{G}_N \). We say that the configuration \( \kappa_0 \) is at time 0, and that four configurations \( \kappa_{4k-3}, \ldots, \kappa_{4k} \) are at time \( k \) for every \( k > 0 \).

Recall that in Section 6.2.2, a global round transition is induced of a sequence of four sub-rounds: Schedule, Send, Receive, and Computation. In an admissible sequence \( \pi = \kappa_0 \kappa_1 \ldots \) of global configurations in \( \mathcal{G}_N \), for every \( k > 0 \), every sub-sequence of four configurations \( \kappa_{4k-3}, \ldots, \kappa_{4k} \) presents one global round transition. Configuration \( \kappa_{4k-3} \) is in sub-round Schedule, and configuration \( \kappa_{4k} \) is in sub-round Computation for every \( k > 0 \). So, the notion of time says that the global round transition \( \kappa_{4k-3} \rightarrow \kappa_{4k} \) happens at time \( k \).

**Message ages.** Now we discuss the formalization of message ages. For every sent message \( m \), the global system tags it with its current age, i.e., \( (m, \text{age}_m) \). Message ages require that the type of message buffers needs to be changed to \( \text{buf} : \mathcal{C}_N \times 1..N \times 1..N \rightarrow \text{Set}(\text{Msg} \times N) \).

In our formalization, when message \( m \) was added to the message buffer in sub-round Send, its age is 0. Instead of predicate \( \text{Sending} \), our formalization now uses the following predicate \( \text{Sending}' \).

\[
\text{Sending}'(\kappa, \kappa', i, m) \triangleq \forall r \in 1..N: (m, 0) \not\in \text{buf}(\kappa, i, r) \\
\land \forall r \in 1..N: \text{buf}(\kappa', i, r) = \{(m, 0)\} \cup \text{buf}(\kappa, i, r) \\
\land \text{lstate}(\kappa, i) \xrightarrow{\text{send}(m)} \text{lstate}(\kappa', i)
\]

Message ages are increased by 1 when the global system takes a \( \xrightarrow{\text{Send}} \) transition. Formally, for every time \( k \geq 0 \), for every process \( s, r \in 1..N \), the following constraints hold:

(i) For every message \( (m, \text{age}_m) \) in \( \text{buf}(\kappa_{4k}, s, r) \), there exists a message \( (m', \text{age}_{m'}) \) in \( \text{buf}(\kappa_{4k+1}, s, r) \) such that \( m = m' \) and \( \text{age}_{m'} = \text{age}_m + 1 \).

(ii) For every message \( (m', \text{age}_{m'}) \) in \( \text{buf}(\kappa_{4k+1}, s, r) \), there exists a message \( (m, \text{age}_m) \) in \( \text{buf}(\kappa_{4k}, s, r) \) such that \( m = m' \) and \( \text{age}_m = \text{age}_{m'} + 1 \).

Constraint (i) ensures that every in-transit message age will be added by one time-unit in the sub-round Schedule. Constraint (ii) ensures that no new messages will be added.
in \( \text{buf}(\kappa_{4k+1}, s, r) \). These two constraints are used to replace Constraint (1d) about unchanged message buffers in the definition of sub-round Schedule in Section 6.2.2.

Moreover, the age of an in-transit message is unchanged in other sub-rounds. Formally, for every time \( k > 0 \), for every \( 0 \leq \ell \leq 3 \), for every pair of processes \( s, r \in 1..N \), for every message \((m, age_m)\) in \( \text{buf}(\kappa_{4k-\ell}, s, r) \), there exists \((m', age_{m'})\) in \( \text{buf}(\kappa_{4k-3}, s, r) \) such that \( m = m' \) and \( age_m = age_{m'} \).

Finally, message ages are not delivered to processes in sub-round Receive. Instead of predicate \( \text{Receiving} \), our formalization now uses the following predicate \( \text{Receiving}' \).

\[
\text{Receiving}(\kappa, \kappa', i, S_1, \ldots, S_N) \triangleq \forall s \in 1..N: S_s \cap \text{buf}(\kappa', s, i) = \emptyset \wedge \forall s \in 1..N: \text{buf}(\kappa', s, i) \cup S_s = \text{buf}(\kappa, s, i) \wedge \text{lstate}(\kappa, i) \xrightarrow{\text{rcr}v(g(S_1), \ldots, g(S_N))} \text{lstate}(\kappa', i)
\]

where function \( g: \text{Set} (\text{Msg} \times \mathbb{N}) \rightarrow \text{Set} (\text{Msg}) \) is to detag message ages in a set \( S \).

Formally, we have two following constraints:

1. For every \((m, age_m)\) \( \in S \), it holds \( m \in g(S) \).
2. For every \( m \in g(S) \), there exists \( age_m \in \mathbb{N} \) such that \((m, age_m)\) \( \in S \).

**Partial synchrony constraints.** We here focus on the case of unknown bounds. Recall that Constraints \([\text{TC1}]\) and \([\text{TC2}]\) hold in this case. Given an admissible sequence \( \pi = \kappa_0 \kappa_1 \ldots \) of global configurations in \( \mathcal{G}_N \), Constraints \([\text{TC1}]\) and \([\text{TC2}]\) on \( \pi \) can be formalized as follows, respectively:

\( \text{(PS1)} \) For every process \( r \in 1..N \), for every time \( k > 0 \), if \( \text{Enabled}(\kappa_{4k-2}, r, \text{Loc}_{rcv}) \), then for every process \( s \in 1..N \), there exists no message \((m, age_m)\) in \( \text{buf}(\kappa_{4k-1}, s, r) \) such that \( age_m \geq \Delta \).

\( \text{(PS2)} \) For every process \( i \in 1..N \), for every time interval \([k, k + \Phi]\), if we have that \( \text{pc}((\text{lstate}(\kappa_t, i)) \neq \ell_{\text{crash}} \) for every configuration index in the time interval \([4k - 3, 4(k + \Phi)]\), then there exist a configuration index \( t \) in \([4k - 3, 4(k + \Phi)]\) and a set \( L \) of locations such that \( \text{Enabled}(\kappa_t, i, L) \) where \( L \) is one of \( \text{Loc}_{\text{snd}}, \text{Loc}_{\text{recv}} \), and \( \text{Loc}_{\text{comp}} \).

Constraint \([\text{PS1}]\) requires that if process \( r \) takes a step from \( \kappa_{4k-2} \) in the sub-round Receive, then there exists no in-transit messages (sent to process \( r \)) whose ages are at least \( \Delta \) time-units in \( \kappa_{4k-1} \), that is, older messages must have been received before that. In principle, partial synchrony allows messages to be older than \( \Delta \) time-units as long as the receiver does not take a step after the message reaches age Delta. Consistent to \([\text{TC1}]\), whenever a receiver takes a step after a message is older than \( \Delta \) time-units, the reception step removes it from the buffer. This limitation is to enabled processes.
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

Constraint [(PS2)] ensures that for every time interval \([k, k + \Phi]\) with configurations \(\kappa_{4k-3}, \ldots, \kappa_{4(k+\Phi)}\), for every process \(i \in 1..N\), if process \(i\) is correct in this time interval, there exist a configuration \(\kappa_{t_0} \in \{\kappa_{4k-3}, \ldots, \kappa_{4(k+\Phi)}\}\) and a set \(L\) of locations such that the location of process \(i\) at \(\kappa_{t_0}\) is in \(L\) and process \(i\) takes a step from \(\kappa_{t_0}\).

Admissible sequences under partial synchrony. Let \(\pi = \kappa_0 \kappa_1 \ldots\) be an admissible sequence of global configurations in \(G_N\). We say that \(\pi\) is under partial synchrony if Constraints [(PS1)] and [(PS2)] hold in \(\pi\). Notice that admissible sequences under partial synchrony allow multiple processes to make a transition in a time unit.

6.3 Cutoff Results in the Model of the Global Distributed Systems

Let \(\mathcal{A}\) be a symmetric point–to–point algorithm. In this section, we show cutoff results for the number of processes in the algorithm \(\mathcal{A}\) in the unrestricted model. These results are Theorems 6.3.1 and 6.3.2, and the detailed proofs are provided in Section 6.4 With these cutoff results, one can verify two properties Strong Completeness and Eventually Strong Accuracy of the failure detector of [CT96] by model checking two instances of sizes 1 and 2 in case of synchrony.

**Theorem 6.3.1** Let \(\mathcal{A}\) be a symmetric point–to–point algorithm under the unrestricted model. Let \(G_1\) and \(G_N\) be instances of 1 and \(N\) processes respectively for some \(N \geq 1\). Let \(Path_1\) and \(Path_N\) be sets of all admissible sequences of configurations in \(G_1\) and in \(G_N\), respectively. Let \(\omega_{\{i\}}\) be an LTL\(\setminus X\) formula in which every predicate takes one of the forms: \(P_1(i)\) or \(P_2(i, i)\) where \(i\) is an index in \(1..N\). Then, it follows that:

\[
(\forall \pi_N \in Path_N: G_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_{\{i\}}) \Leftrightarrow (\forall \pi_1 \in Path_1: G_1, \pi_1 \models \omega_{\{1\}})
\]

**Theorem 6.3.2** Let \(\mathcal{A}\) be a symmetric point–to–point algorithm under the unrestricted model. Let \(G_2\) and \(G_N\) be instances of 2 and \(N\) processes respectively for some \(N \geq 2\). Let \(Path_2\) and \(Path_N\) be sets of all admissible sequences of configurations in \(G_2\) and in \(G_N\), respectively. Let \(\psi_{\{i,j\}}\) be an LTL\(\setminus X\) formula in which every predicate takes one of the forms: \(P_1(i)\), or \(P_2(j)\), or \(P_3(i, j)\), or \(P_4(j, i)\) where \(i\) and \(j\) are different indexes in \(1..N\). It follows that:

\[
(\forall \pi_N \in Path_N: G_N, \pi_N \models \bigwedge_{i \neq j}^{i,j \in 1..N} \psi_{\{i,j\}}) \Leftrightarrow (\forall \pi_2 \in Path_2: G_2, \pi_2 \models \psi_{\{1,2\}})
\]

Since the proof of Theorem 6.3.1 is similar to the one of Theorem 6.3.2, we focus on Theorem 6.3.2 here. Its proof is based on the symmetric characteristics in the system model (the network topology and the three functions \(nextLoc, genMsg\), and \(nextVar\)) and correctness properties, and on the following lemmas.
The application of a transposition on a set of indexes 1..N preserves the structure of the process template \( U_N \).

- Lemma 6.3.2 says that every transposition on a set of indexes 1..N preserves the structure of the global transition system \( G_N \) for every \( N \geq 1 \).

- Lemma 6.3.3 says that \( G_2 \) and \( G_N \) are trace equivalent under a set \( AP_{\{1,2\}} \) of predicates that take one of the forms: \( P_1(i) \), or \( P_2(j) \), or \( P_3(i,j) \), or \( P_4(j,i) \).

In the following, we present definitions and constructions to prove these lemmas.

### 6.3.1 Index Transpositions and Symmetric Point-to-point Systems

We first recall the definition of transposition. Given a set 1..N of indexes, we call a bijection \( \alpha: 1..N \rightarrow 1..N \) a transposition between two indexes \( i, j \in 1..N \) if the following properties hold: \( \alpha(i) = j \), and \( \alpha(j) = i \), and \( \forall k \in 1..N: (k \neq i \land k \neq j) \Rightarrow \alpha(k) = k \). We let \((i \leftrightarrow j)\) denote a transposition between two indexes \( i \) and \( j \).

The application of a transposition to a template state is given in Definition 6.3.1. Informally, applying a transposition \( \alpha = (i \leftrightarrow j) \) to a template state \( \rho \) generates a new template state by switching only the evaluation of \( rcd \) and \( lvar \) at indexes \( i \) and \( j \). The application of a transposition to a global configuration is provided in Definition 6.3.2.

In addition to process configurations, we need to change message buffers and control components. We override notation by writing \( \alpha_S(\rho) \) and \( \alpha_C(\kappa) \) to refer the application of a transposition \( \alpha \) to a state \( \rho \) and to a configuration \( \kappa \), respectively. These functions \( \alpha_Q \) and \( \alpha_C \) are named a local transposition and a global transposition, respectively.

**Definition 6.3.1 (Local Transposition)** Let \( U_N \) be a process template with process indexes 1..N, and \( \rho = (\ell, S_1, \ldots, S_N, d_1, \ldots, d_N) \) be a state in \( U_N \). Let \( \alpha = (i \leftrightarrow j) \) be a transposition on 1..N. The application of \( \alpha \) to \( \rho \), denoted as \( \alpha_S(\rho) \), generates a tuple \( (\ell', S'_1, \ldots, S'_N, d'_1, \ldots, d'_N) \) such that

1. \( \ell = \ell' \), and \( S_i = S'_i \), and \( S_j = S'_j \), and \( d_i = d'_j \) and \( d_j = d'_i \), and
2. \( \forall k \in 1..N: (k \neq i \land k \neq j) \Rightarrow (S_k = S'_k \land d_k = d'_k) \)

**Definition 6.3.2 (Global Transposition)** Let \( G_N \) be a global system with process indexes 1..N, and \( \kappa \) be a configuration in \( G_N \). Let \( \alpha = (i \leftrightarrow j) \) be a transposition on 1..N. The application of \( \alpha \) to \( \kappa \), denoted as \( \alpha_C(\kappa) \), generates a configuration in \( G_N \) which satisfies following properties:

1. \( \forall i \in 1..N: Istate(\alpha_C(\kappa), \alpha(i)) = Istate(\kappa, i) \).
2. \( \forall s, r \in 1..N: Buf(\alpha_C(\kappa), \alpha(s), \alpha(r)) = Buf(\kappa, s, r) \)
3. \( \forall i \in 1..N: Active(\alpha_C(\kappa), \alpha(i)) = Active(\kappa, i) \)
Since the content of every message in $\Msg$ does not have indexes of the receiver and sender, no transposition affects the messages. We define the application of a transposition to one of send, compute, crash, and stutter template transitions return the same transition. We extend the application of a transposition to a receive template transition as in Definition 6.3.3.

**Definition 6.3.3 (Receive-transition Transposition)**  Let $\mathcal{U}_N$ be a process template with indexes $1..N$, and $\alpha = (i \leftrightarrow j)$ be a transposition on $1..N$. Let $\crev(S_1, ..., S_N)$ be a transition in $\mathcal{U}_N$ which refers to a task with a receive action. We let $\alpha_{\crev}(\crev(S_1, ..., S_N))$ denote the application of $\alpha$ to $\crev(S_1, ..., S_N)$, and this application returns a new transition $\crev(S'_1, ..., S'_N)$ in $\mathcal{U}_N$ such that:

1. $S_i = S'_j$ and $S_j = S'_i$, and
2. $\forall k \in 1..N$: $(k \neq i \land k \neq j) \Rightarrow (S_k = S'_k \land d_k = d'_k)$

We let $\alpha_{U}(\mathcal{U}_N)$ and $\alpha_{C}(\mathcal{G}_N)$ denote the application of a transposition $\alpha$ to a process template $\mathcal{U}_N$ and a global transition system $\mathcal{G}_N$, respectively. Since these definitions are straightforward, we skip them in this chapter. We prove later that $\alpha_{S}(\mathcal{U}_N) = \mathcal{U}_N$ and $\alpha_{C}(\mathcal{G}_N) = \mathcal{G}_N$ (see Lemmas 6.3.1 and 6.3.2).

**Lemma 6.3.1 (Symmetric Process Template)**  Let $\mathcal{U}_N = (Q_N, \mathcal{T}_N, \mathcal{R}_N, q^0_N)$ be a process template with indexes $1..N$. Let $\alpha = (i \leftrightarrow j)$ be a transposition on $1..N$, and $\alpha_{\mathcal{Q}}$ be a local transposition based on $\alpha$ (from Definition 6.3.1). The following properties hold:

1. $\alpha_{\mathcal{Q}}$ is a bijection from $Q_N$ to itself.
2. The initial state is preserved under $\alpha_{\mathcal{Q}}$, i.e. $\alpha_{S}(q^0_N) = q^0_N$.
3. Let $\rho, \rho' \in \mathcal{U}_N$ be states such that $\rho \xrightarrow{\crev(S_1, ..., S_N)} \rho'$ for some sets of messages $S_1, ..., S_N$ in $\mathcal{S}(\Msg)$. It follows $\alpha_{S}(\rho) \xrightarrow{\alpha_{\crev}(\crev(S_1, ..., S_N))} \alpha_{S}(\rho')$.
4. Let $\rho, \rho'$ be states in $\mathcal{U}_N$, and $\mathcal{tr} \in \mathcal{T}_N$ be one of send, local computation, crash and stutter transitions such that $\rho \xrightarrow{\mathcal{tr}} \rho'$. Then, $\alpha_{S}(\rho) \xrightarrow{\mathcal{tr}} \alpha_{S}(\rho')$.

**Lemma 6.3.2 (Symmetric Global System)**  Let $\mathcal{G}_N = (\mathcal{C}_N, \mathcal{T}_N, \mathcal{R}_N, q^0_N)$ be a global transition system. Let $\alpha$ be a transposition on $1..N$, and $\alpha_{\mathcal{C}}$ be a global transposition based on $\alpha$ (from Definition 6.3.2). The following properties hold:

1. $\alpha_{\mathcal{C}}$ is a bijection from $\mathcal{C}_N$ to itself.
2. The initial configuration is preserved under $\alpha_{\mathcal{C}}$, i.e. $\alpha_{\mathcal{C}}(q^0_N) = q^0_N$.
3. Let $\kappa$ and $\kappa'$ be configurations in $\mathcal{G}_N$, and $\mathcal{tr} \in \mathcal{T}_N$ be either an internal transition such that $\kappa \xrightarrow{\mathcal{tr}} \kappa'$. It follows $\alpha_{\mathcal{C}}(\kappa) \xrightarrow{\mathcal{tr}} \alpha_{\mathcal{C}}(\kappa')$.
4. Let $\kappa$ and $\kappa'$ be configurations in $\mathcal{G}_N$. If $\kappa \sim \kappa'$, then $\alpha_{\mathcal{C}}(\kappa) \sim \alpha_{\mathcal{C}}(\kappa')$. 
6.3.2 Trace Equivalence of $\mathcal{G}_2$ and $\mathcal{G}_N$ under $AP_{\{1,2\}}$

Let $\mathcal{G}_2$ and $\mathcal{G}_N$ be two global transition systems whose processes follow the same symmetric point–to–point algorithm. In the following, our goal is to prove Lemma 6.3.5 that says $\mathcal{G}_2$ and $\mathcal{G}_N$ are trace equivalent under a set $AP_{\{1,2\}}$ of predicates which take one of the forms: $Q_1(1)$, $Q_2(2)$, $Q_3(1,2)$, or $Q_4(2,1)$. To do that, we first present two construction techniques: Construction 6.3.1 to construct a state in $\mathcal{U}_2$ from a state in $\mathcal{U}_N$, and Construction 6.3.2 to construct a global configuration in $\mathcal{G}_2$ from a given global configuration in $\mathcal{G}_N$. Second, we define trace equivalence under a set $AP_{\{1,2\}}$ of predicates which take one of the forms: $P_1(i)$, or $P_2(j)$, or $P_3(i,j)$, or $P_4(j,i)$. Our definition of trace equivalence under $AP_{\{1,2\}}$ is extended from the definition of trace equivalence in [Hoa80].

Next, we present two Lemmas 6.3.3 and 6.3.4. These lemmas are required in the proof of Lemma 6.3.5.

To keep the presentation simple, when the context is clear, we simply write $U_N$, instead of fully $U_N = (Q_N, Tr_N, Rel_N, q^0_N)$. We also write $G_N$, instead of fully $G_N = (C_N, T_N, R_N, g^0_N)$.

Construction 6.3.1 (State Projection) Let $\mathcal{A}$ be an arbitrary symmetric point–to–point algorithm. Let $U_N$ be a process template of $\mathcal{A}$ for some $N \geq 2$, and $\rho^N$ be a process configuration of $U_N$. We construct a tuple $\rho^2 = (pc_1, rcvd_1, rcvd_2, v_1, v_2)$ based on $\rho^N$ and a set $\{1, 2\}$ of process indexes in the following way:

1. $pc_1 = pc(\rho^N)$.
2. For every $i \in \{1, 2\}$, it follows $rcvd_i = rcvd(\rho^N, i)$.
3. For every $i \in \{1, 2\}$, it follows $v_i = lvar(\rho^N, i)$.

Construction 6.3.2 (Configuration Projection) Let $\mathcal{A}$ be a symmetric point–to–point algorithm. Let $\mathcal{G}_2$ and $\mathcal{G}_N$ be two global transition systems of two instances of $\mathcal{A}$ for some $N \geq 2$, and $\kappa^N \in C_N$ be a global configuration in $\mathcal{G}_N$. A tuple

$$\kappa^2 = (s_1, s_2, buf_1^1, buf_2^1, buf_1^2, buf_2^2, act_1, act_2)$$

is constructed based on the configuration $\kappa^N$ and a set $\{1, 2\}$ of indexes in the following way:

1. For every $i \in \{1, 2\}$, a component $s_i$ is constructed from $lstate(\kappa^N, i)$ with Construction 6.3.1 and indexes $\{1, 2\}$.
2. For every $s, r \in \{1, 2\}$, it follows $buf_s^r = buf(\kappa^N, s, r)$.
3. For every process $i \in \{1, 2\}$, it follows $act_i = active(\kappa^N, i)$.
6. CUTOFFS FOR SYMMETRIC POINT-TO-POINT DISTRIBUTED ALGORITHMS

Figure 6.1: Given execution in \( G_3 \), construct an execution in \( G_2 \) by index projection.

Note that a tuple \( \rho^2 \) constructed with Construction 6.3.1 is a state in \( U_2 \), and a tuple \( \kappa^2 \) constructed with Construction 6.3.2 is a configuration in \( G_2 \). We call \( \rho^2 \) (and \( \kappa^2 \)) the index projection of \( \rho^N \) (and \( \kappa^N \)) on indexes \{1, 2\}. The following Lemma 6.3.3 says that Construction 6.3.2 allows us to construct an admissible sequence of global configurations in \( G_2 \) based on a given admissible sequence in \( G_N \). Intuitively, the index projection throws away processes \( 3..N \) as well as their corresponding messages and buffers. Moreover, for every \( i, j \in \{1, 2\} \), the index projection preserves (i) when process \( i \) takes a step, and (ii) what action process \( i \) takes at time \( t \geq 0 \), and (iii) messages between process \( i \) and process \( j \). For example, Figure 6.1 demonstrates an execution in \( G_2 \) that is constructed based on a given execution in \( G_3 \) with the index projection.

Lemma 6.3.3 Let \( A \) be a symmetric point–to–point algorithm. Let \( G_2 \) and \( G_N \) be two transition systems such that all processes in \( G_2 \) and \( G_N \) follow \( A \), and \( N \geq 2 \). Let \( \pi^N = \kappa^N_0, \kappa^N_1, \ldots \) be an admissible sequence of configurations in \( G_N \). Let \( \pi^2 = \kappa^2_0, \kappa^2_1, \ldots \) be a sequence of configurations in \( G_2 \) such that \( \kappa^2_k \) is the index projection of \( \kappa^N_k \) on indexes \{1, 2\} for every \( k \geq 0 \). Then, \( \pi^2 \) is admissible in \( G_2 \).

Proof. [Sketch of proof] The proof of Lemma 6.3.3 is based on the following observations:

1. The application of Construction 6.3.1 to an initial template state of \( U_N \) constructs an initial template state of \( U_2 \).
2. Construction 6.3.1 preserves the template transition relation.
3. The application of Construction 6.3.2 to an initial global configuration of \( G_N \) constructs an initial global configuration of \( G_2 \).
4. Construction 6.3.2 preserves the global transition relation.

Moreover, Lemma 6.3.4 says that given an admissible sequence \( \pi^2 = \kappa^2_0, \kappa^2_1, \ldots \) in \( G_2 \), there exists an admissible sequence \( \pi^N = \kappa^N_0, \kappa^N_1, \ldots \) in \( G_N \) such that \( \kappa^N_i \) is the index projection of \( \kappa^N_i \) on indexes \{1, 2\} for every \( 0 \leq i \).

Definition 6.3.4 (Trace Equivalence under \( AP_{\{1,2\}} \)). Let \( A \) be an arbitrary symmetric point–to–point algorithm. Let \( G_2 = (Q_2, T_2, Rel_2, q^0_2) \) and \( G_N = (Q_N, T_N, Rel_N, q^0_N) \) be
global transition systems of $A$ for some $N \geq 2$. Let $AP_{\{1,2\}}$ be a set of predicates that take one of the forms: $P_1(i)$, or $P_2(j)$, or $P_3(i,j)$, or $P_4(j,i)$. Let $L: Q_2 \cup Q_N \rightarrow 2^{AP}$ be an evaluation function. We say that $G_2$ and $G_N$ are trace equivalent under $AP_{\{1,2\}}$ if the following constraints hold:

1. For every admissible sequence $\pi^2 = \kappa^2_0 \kappa^2_1 \ldots$ of configurations in $G_2$, there exists an admissible sequence of configurations $\pi^N = \kappa^N_0 \kappa^N_1 \ldots$ in $G_N$ such that $L(\kappa^2_i) = L(\kappa^N_i)$ for every $i \geq 0$.

2. For every admissible sequence $\pi^N = \kappa^N_0 \kappa^N_1 \ldots$ in $G_N$, there exists an admissible sequence $\pi^2 = \kappa^2_0 \kappa^2_1 \ldots$ of configurations in $G_2$ such that $L(\kappa^2_i) = L(\kappa^N_i)$ for every $i \geq 0$.

Lemma 6.3.4 Let $A$ be an arbitrary symmetric point–to–point algorithm. Let $G_2$ and $G_N$ be global transition systems of $A$ for some $N \geq 2$. Let $\pi^2 = \kappa^2_0 \kappa^2_1 \ldots$ be an admissible sequence of configurations in $G_2$. There exists an admissible sequence $\pi^N = \kappa^N_0 \kappa^N_1 \ldots$ of configurations in $G_N$ such that $\kappa^2_i$ is the index projection of $\kappa^N_i$ on indexes $\{1, 2\}$ for every $i \geq 0$.

Sketch of proof. We construct an execution $\pi_N$ in $G_N$ based on $\pi_2$ such that all processes $3..N$ crash from the beginning, and $\pi_2$ is an index projection of $\pi_N$. For instance, Figure 6.2 demonstrates an execution in $G_3$ that is constructed based on one in $G_2$. We have that $\pi_2$ is admissible in $G_2$.

Lemma 6.3.5 Let $A$ be a symmetric point–to–point algorithm. Let $G_2$ and $G_N$ be its instances for some $N \geq 2$. Let $AP_{\{1,2\}}$ be a set of predicates that take one of the forms: $P_1(1)$, $P_2(2)$, $P_3(1,2)$ or $P_4(2,1)$. It follows that $G_2$ and $G_N$ are trace equivalent under $AP_{\{1,2\}}$.

Sketch of proof. The proof of Lemma 6.3.5 is based on Definition 6.3.4 Lemma 6.3.3 and Lemma 6.3.4.
6.4 Detailed Proofs for Cutoff Results in the Model of the Global Distributed Systems

In this section, we present the detailed proofs for Theorems 6.3.1 and 6.3.2. In Sections 6.4.1 and 6.4.2, we prove that every transposition on a set of process indexes 1..N preserves the structure of the process template \( \mathcal{U}_N \) and the structure of the global transition system \( \mathcal{G}_N \) for every \( N \geq 1 \), respectively. In Section 6.4.3, we show that \( \mathcal{G}_2 \) and \( \mathcal{G}_N \) are trace equivalent under \( AP_{\{1,2\}} \). Next, we prove that \( \mathcal{G}_1 \) and \( \mathcal{G}_N \) are trace equivalent under \( AP_{\{1\}} \) in Section 6.4.4. Then, the detailed proofs for Theorems 6.3.1 and 6.3.2 is presented in Section 6.4.5. Finally, we discuss why we can verify the strong completeness property of the failure detector of \( CT96 \) under synchrony by model checking instances of size 2 by applying our cutoff results.

6.4.1 Transpositions and Process Templates

The proof of Lemma 6.3.1 requires the following propositions:

- Given a transposition \( \alpha \), Proposition 6.4.1 says that a function \( \alpha_q \), which refers to the application of \( \alpha \) to a state in \( Q_N \), is a bijection from \( Q_N \) to itself.
- Proposition 6.4.2 says that \( \alpha_q \) has no effect on the initial template state \( q_0^N \).
- Propositions 6.4.3 and 6.4.4 describe the relationship between transpositions and template transitions.

Lemma 6.3.1. Let \( \mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_0^N) \) be a process template with indexes 1..N. Let \( \alpha = (i \leftrightarrow j) \) be a transposition on 1..N, and \( \alpha_q \) be a local transposition based on \( \alpha \) (from Definition 6.3.1). The following properties hold:

1. \( \alpha_q \) is a bijection from \( Q_N \) to itself.
2. The initial state is preserved under \( \alpha_q \), i.e., \( \alpha_S(q_0^N) = q_0^N \).
3. Let \( \rho, \rho' \in \mathcal{U}_N \) be states such that \( \rho \xrightarrow{cr.ev(S_1,\ldots,S_N)} \rho' \) for some sets of messages \( S_1,\ldots,S_N \in \text{Set}(Msg) \). It follows that \( \alpha_S(\rho) \xrightarrow{\alpha_r(cr.ev(S_1,\ldots,S_N))} \alpha_S(\rho') \).
4. Let \( \rho, \rho' \) be states in \( \mathcal{U}_N \), and \( tr \in Tr_N \) be one of send, local computation, crash and stutter transitions such that \( \rho \xrightarrow{tr} \rho' \). Then, \( \alpha_S(\rho) \xrightarrow{tr} \alpha_S(\rho') \).

Proof. We have: point 1 holds by Proposition 6.4.1 and point 2 holds by Proposition 6.4.2 and point 3 holds by Proposition 6.4.3 and point 4 holds by Proposition 6.4.4.

Proposition 6.4.1 Let \( \mathcal{U}_N = (Q_N, Tr_N, Rel_N, q_0^N) \) be a process template with indexes 1..N. Let \( \alpha = (i \leftrightarrow j) \) be a transposition on 1..N, and \( \alpha_q \) be a local transposition based on \( \alpha \) (from Definition 6.3.1). Then, \( \alpha_q \) is a bijection from \( Q_N \) to itself.
6.4. Detailed Proofs for Cutoff Results in the Model of the Global Distributed Systems

Proof. Since two transpositions \((i \leftrightarrow j)\) and \((j \leftrightarrow i)\) are equivalent, we assume \(i < j\). To show that \(\alpha_Q\) is a bijection from \(Q_N\) to itself, we prove that the following properties hold:

1. For every template state \(\rho' \in Q_N\), there exists a template configuration \(\rho \in Q_N\) such that \(\alpha_S(\rho) = \rho'\).
2. For every pair of states \(\rho_1, \rho_2 \in Q_N\), if \(\alpha_S(\rho_1) = \alpha_S(\rho_2)\), then \(\rho_1 = \rho_2\).

We first show that Point 1 holds. Assume that \(\rho' \in Q_N\) is a following tuple

\[
\rho' = (\ell, S_1, \ldots, S_i, \ldots, S_j, \ldots, S_N, d_1, \ldots, d_i, \ldots, d_j, \ldots, d_N)
\]

where \(\ell \in \text{Loc}, S_i \in \text{Set}(\text{msg}), d_i \in \mathcal{D}\) for every \(i \in 1..N\). Let \(\rho\) be the following tuple

\[
\rho = (\ell, S_1, \ldots, S_i, \ldots, S_j, \ldots, S_N, d_1, \ldots, d_i, \ldots, d_j, \ldots, d_N)
\]

where \(S_k = S_k' \land d_k = d_k'\) for every \(k \in 1..N \setminus \{i, j\}\). By Definition 6.3.1, we have \(\alpha_S(\rho) = \rho'\). Moreover, by the definition of a process template in Section 6.2.1, it follows \(\rho \in Q_N\).

We now focus on Point 2. By definition of the application of a process-index transposition to a template state, it is easy to check that \(\alpha_Q((\alpha_S(\rho_1))) = \rho\) for every \(\rho \in Q_N\). It follows that \(\rho_1 = \alpha_Q((\alpha_S(\rho_1))) = \alpha_Q((\alpha_S(\rho_2))) = \rho_2\) since \(\alpha_S(\rho_1) = \alpha_S(\rho_2)\).

Therefore, Proposition 6.4.1 holds.

**Proposition 6.4.2** Let \(U_N = (Q_N, Tr_N, Rel_N, q^0_N)\) be a process template. Let \(\alpha = (i \leftrightarrow j)\) be a transposition on 1..N, and \(\alpha_Q\) be a local transposition based on \(\alpha\) (from Definition 6.3.1). It follows that \(\alpha_S(q^0_N) = q^0_N\).

Proof. By definition of \(q^0_N\) in Section 6.2.1, we have \(\text{rcvd}(q^0_N, i) = \text{rcvd}(q^0_N, j)\) and \(\text{lvar}(q^0_N, i) = \text{lvar}(q^0_N, j)\). It immediately follows \(\alpha_S(q^0_N) = q^0_N\).

**Proposition 6.4.3** Let \(U_N = (Q_N, Tr_N, Rel_N, q^0_N)\) be a process template with indexes 1..N. Let \(\rho_0\) and \(\rho_1\) be states in \(U_N\) such that \(\rho_0 \xrightarrow{\text{crce}(S_1, \ldots, S_N)} \rho_1\) for some sets of messages: \(S_1, \ldots, S_N \subseteq \text{Set}(\text{msg})\). Let \(\alpha = (i \leftrightarrow j)\) be a transposition on 1..N, and \(\alpha_R\) be a receive-transition transposition based on \(\alpha\) (from Definition 6.3.3). It follows \(\alpha_S(\rho_0) \xrightarrow{\alpha_R(\text{crce}(S_1, \ldots, S_N))} \alpha_S(\rho_1)\).

Proof. We prove that all Constraints (a)–(c) between two states \(\alpha_S(\rho_0)\) and \(\alpha_S(\rho_1)\) in the transition \(\text{csnd}\) defined in Section 6.2.1 hold. First, we focus on Constraint (a). We have \(\text{pc}(\alpha_S(\rho_1)) = \text{pc}(\rho_1)\) by Definition 6.3.1. We have \(\text{pc}(\rho_1) = \text{nextLoc}(\text{pc}(\rho_0))\) by the semantics of \(\text{crce}(S_1, \ldots, S_N)\) in Section 6.2.1. We have \(\text{nextLoc}(\text{pc}(\rho_0)) = \text{nextLoc}(\text{pc}(\alpha_S(\rho_0)))\) by Definition 6.3.1. It follows \(\text{pc}(\alpha_S(\rho_1)) = \text{nextLoc}(\text{pc}(\alpha_S(\rho_0)))\).

Moreover, we have \(\{m\} = \text{genMsg}(\text{pc}(\rho_0))\) (by the semantics of \(\text{crce}\) in Section 6.2.1)
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

\[ = \text{genMsg}(pc(\alpha_S(@0))) \] (by Definition 6.3.1)

Hence, Constraint (a) holds.

Now we focus on Constraint (b). By Definition 6.3.1, we have \(\text{rcvd}(\alpha_S(@0), k) = \text{rcvd}(@0, k)\) and \(\text{rcvd}(\alpha_S(@1), k) = \text{rcvd}(@1, k)\) for every \(k \in 1..N \setminus \{i, j\}\). We have \(\text{rcvd}(@1, k) = S_k \cup \text{rcvd}(@0, k)\) by the semantics of \(\text{rcrv}(S_1, \ldots, S_N)\) in Section 6.2.1. It follows \(\text{rcvd}(\alpha_S(@1), k) = S_k \cup \text{rcvd}(\alpha_S(@0), k)\) for every \(k \in 1..N \setminus \{i, j\}\). Now we focus on \(\text{rcvd}(\alpha_S(@1), i)\). We have \(\text{rcvd}(\alpha_S(@1), i) = \text{rcvd}(@1, j)\) and \(\text{rcvd}(\alpha_S(@0), i) = \text{rcvd}(@0, j)\) by Definition 6.3.1. Since \(\text{rcvd}(@1, j) = \text{rcvd}(@0, j) \cup S_j\) it follows \(\text{rcvd}(\alpha_S(@1), i) = \text{rcvd}(\alpha_S(@0), i) \cup S_j\). By similar arguments, we have \(\text{rcvd}(\alpha_S(@1), j) = \text{rcvd}(\alpha_S(@0), j) \cup S_i\). Hence, Constraint (b) holds.

Now we focus on Constraint (c). By similar arguments in the proof of Constraint (b), for every \(k \in 1..N \setminus \{i, j\}\), we have

\[ \text{lvar}(\alpha_S(@1), k) = \text{nextVar}(pc(\alpha_S(@0)), S_k, \text{lvar}(\alpha_S(@0), k)) \]

Now we focus on \(\text{lvar}(\alpha_S(@1), i)\). We have \(\text{lvar}(\alpha_S(@1), i) = \text{lvar}(@1, j)\) by Definition 6.3.1. By the semantics of \(\text{rcrv}(S_1, \ldots, S_N)\) in Section 6.2.1, it follows that \(\text{lvar}(@1, j) = \text{nextVar}(pc(@0), S_j, \text{lvar}(@0, j))\). By Definition 6.3.1, we have

\[
\begin{align*}
\text{lvar}(\alpha_S(@1), i) \\
= \text{lvar}(@1, j) \\
= \text{nextVar}(pc(@0), S_j, \text{lvar}(@0, j)) \\
= \text{nextVar}(pc(\alpha_S(@0)), S_j, \text{lvar}(\alpha_S(@0), i))
\end{align*}
\]

Moreover, by similar arguments, we have

\[ \text{lvar}(\alpha_S(@1), j) = \text{nextVar}(pc(\alpha_S(@0)), S_i, \text{lvar}(\alpha_S(@0), i)) \]

Constraint (c) holds. Hence, we have \(\alpha_S(@0) \xrightarrow{\alpha_S(\text{rcrv}(S_1, \ldots, S_N))} \alpha_S(@1)\).

**Proposition 6.4.4** Let \(\mathcal{U}_N = (Q_N, Tr_N, \text{Rel}_N, \delta_N^0)\) be a process template with indexes \(1..N\). Let \(\rho \) and \(\rho’\) be states in \(\mathcal{U}_N\), and \(tr \in Tr_N\) be a transition such that \(\rho \xrightarrow{tr} \rho’\) and \(tr\) refers to a task without a receive action. Let \(\alpha = (i \leftrightarrow j)\) be a transposition on \(1..N\), and \(\alpha_Q\) be a local transposition based on \(\alpha\) (from Definition 6.3.1). It follows \(\alpha_S(\rho) \xrightarrow{tr} \alpha_S(\rho’).\)

**Proof.** We prove Proposition 6.4.4 by case distinction.

- **Case \(\rho_0 \xrightarrow{\text{send}(m)} \rho_1\).** By similar arguments in the proof of Proposition 6.4.3, it follows \(pc(\alpha_S(@1)) = \text{nextLoc}(pc(\alpha_S(@0)))\) and \(\{m\} = \text{genMsg}(pc(\alpha_S(@0)))\). Constraint (a) holds. By Definition 6.3.1, for every \(k \in 1..N \setminus \{i, j\}\), we have
The proof strategy of Lemma 6.3.2 is similar to the one of Lemma 6.3.1, and the proof of Proposition 6.4.4 holds.

Hence, Proposition 6.4.4 holds.

6.4.2 Transpositions and Global Systems

The proof strategy of Lemma 6.3.2 is similar to the one of Lemma 6.3.1, and the proof of Lemma 6.3.2 requires the following Propositions 6.4.5, 6.4.6, 6.4.7, and 6.4.8.

Lemma 6.3.2.

Let $G_N = (C_N, T_N, R_N, g_N^0)$ be a global transition system. Let $\alpha$ be a transposition on 1..N, and $\alpha_C$ be a global transposition based on $\alpha$ (from Definition 6.3.2). The following properties hold:

1. $\alpha_C$ is a bijection from $C_N$ to itself.
2. The initial configuration is preserved under $\alpha_C$, i.e. $\alpha_C(g_N^0) = g_N^0$.
3. Let $\kappa$ and $\kappa'$ be configurations in $G_N$, and $tr \in T_N$ be either an internal transition such that $\kappa \xrightarrow{tr} \kappa'$. It follows $\alpha_C(\kappa) \xrightarrow{tr} \alpha_C(\kappa')$.
4. Let $\kappa$ and $\kappa'$ be configurations in $G_N$. If $\kappa \sim \kappa'$, then $\alpha_C(\kappa) \sim \alpha_C(\kappa')$.

Proof. We have: point 1 holds by Proposition 6.4.5 and point 2 holds by Proposition 6.4.6, and point 3 holds by Proposition 6.4.7, and point 4 holds by Proposition 6.4.8. □
Proposition 6.4.5 Let $G_N = (C_N, T_N, R_N, g^0_N)$ be a global transition system with indexes $1..N$. Let $\alpha$ be a process–index transposition on $1..N$, and $\alpha_C$ be a global transposition based on $\alpha$ (from Definition 6.3.2). Then, $\alpha_C$ is a bijection from $C_N$ to itself.

Proof. By applying similar arguments in the proof of Proposition 6.4.1.

Proposition 6.4.6 Let $G_N = (C_N, T_N, R_N, g^0_N)$ be a global transition system with indexes $1..N$. Let $\alpha$ be a process–index transposition on $1..N$, and $\alpha_C$ be a global transposition based on $\alpha$ (from Definition 6.3.2). It follows that $\alpha_C(g^0_N) = g^0_N$.

Proof. By applying similar arguments in the proof of Proposition 6.4.2.

Proposition 6.4.7 Let $G_N = (C_N, T_N, R_N, g^0_N)$ be a global transition system with indexes $1..N$ and a process template $U_N = (Q_N, T_N, R_N, q^0_N)$. Let $\alpha$ be a process–index transposition on $1..N$, and $\alpha_C$ be a global transposition based on $\alpha$ (from Definition 6.3.2). Let $\kappa$ and $\kappa'$ be configurations in $G_N$, and $tr \in T_N$ be an internal transition such that $\kappa \xrightarrow{tr} \kappa'$. Then, $\alpha_C(\kappa) \xrightarrow{tr} \alpha_C(\kappa')$.

Proof. We prove Proposition 6.4.7 by case distinction.

1. Sub-round Schedule. We prove that all Constraints (a)–(e) for the sub-round Schedule hold as follows.

   First, we focus on Constraint (a). By Proposition 6.4.5, both $\alpha_C(\kappa)$ and $\alpha_C(\kappa')$ are configurations in $G_N$. By Definition 6.3.2, for every $i \in 1..N$, we have $\text{active}(\alpha_C(\kappa), \alpha(i)) = \text{active}(\kappa, i)$. We have $\neg\text{active}(\kappa, i)$ by the semantics of the sub-round Schedule in Section 6.2.2. It follows $\neg\text{active}(\alpha_C(\kappa), \alpha(i))$. Hence, the sub-round Schedule can start with a configuration $\alpha_C(\kappa_0)$. Constraint (a) holds.

   Now we focus on Constraint (b) by examining process transitions. For every $i \in 1..N$, by Lemma 6.3.1, we have

   $$lstate(\kappa, i) \xrightarrow{\text{stutter}} lstate(\kappa', i) \Rightarrow \alpha_S(lstate(\kappa, i)) \xrightarrow{\text{stutter}} \alpha_S(lstate(\kappa', i))$$

   By Definition 6.3.2, it follows

   $$\alpha_S(lstate(\kappa, i)) = lstate(\alpha_C(\kappa), \alpha(i))$$
   $$\alpha_S(lstate(\kappa', i)) = lstate(\alpha_C(\kappa'), \alpha(i))$$

   Hence, it follows

   $$lstate(\kappa, i) \xrightarrow{\text{stutter}} lstate(\kappa', i) \Rightarrow lstate(\alpha_C(\kappa), \alpha(i)) \xrightarrow{\text{stutter}} lstate(\alpha_C(\kappa'), \alpha(i))$$

   134
By similar arguments, we have

\[
\text{lstate}(\kappa, i) \xrightarrow{\text{crash}} \text{lstate}(\kappa', k)
\]

\[
\Rightarrow \text{lstate}(\alpha_C(\kappa), \alpha(i)) \xrightarrow{\text{crash}} \text{lstate}(\alpha_C(\kappa'), \alpha(k))
\]

Hence, every process makes either a crash transition or a stuttering step from a configuration \(\alpha_C(\kappa)\) to a configuration \(\alpha_C(\kappa')\). Constraint (b) holds.

We now focus on Constraint (c) by examining control components of crashed processes. Assume that \(pc(\text{lstate}(\kappa', r)) = \ell_{\text{crash}}\) for some \(i \in 1..N\). By the semantics of the sub-round Schedule in Section 6.2.2, we have \(\neg\text{active}(\kappa', i)\). By Definition 6.3.2, it follows \(\neg\text{active}(\alpha_C(\kappa), \alpha(i)) \land \neg\text{active}(\kappa', i)\). Therefore, Constraint (c) holds.

Now we focus on Constraint (d) by examining incoming message buffers to correct processes. By Definition 6.3.2, we have \(\text{buf}(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = \text{buf}(\kappa', s, r)\) for every \(s, r \in 1..N\). By the semantic of the sub-round Schedule in Section 6.2.2, if \(pc(\text{lstate}(\kappa', r)) \neq \ell_{\text{crash}}, \text{ then } \text{buf}(\kappa', s, r) = \text{buf}(\kappa, s, r)\). By Definition 6.3.2, we have \(\text{buf}(\kappa, s, r) = \text{buf}(\alpha_C(\kappa), \alpha(s), \alpha(r))\). Hence, Constraint (d) holds since \(\text{buf}(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = \text{buf}(\alpha_C(\kappa), \alpha(s), \alpha(r))\)

Now we focus on Constraint (e) by examining incoming message buffers to a crashed process. Let \(r\) be an index in \(1..N\) such that \(pc(\text{lstate}(\kappa', r)) = \ell_{\text{crash}}\). By similar arguments in the above case of \(pc(\text{lstate}(\kappa', r)) \neq \ell_{\text{crash}}, \text{ we have that if } pc(\text{lstate}(\kappa', r)) = \ell_{\text{crash}}, \text{ then } \text{buf}(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = \text{buf}(\kappa', s, r)\). By the semantics of the sub-round Schedule in Section 6.2.2, we have \(\text{buf}(\kappa', s, r) = \emptyset\). It follows that \(\text{buf}(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = \emptyset\). Therefore, Constraint (e) holds.

It implies \(\alpha_C(\kappa) \xrightarrow{\text{Sched}} \alpha_C(\kappa')\).

2. Sub-round Send. We prove that all Constraints (a)–(c) for the sub-round Send hold as follows. By Definition 6.3.2, we have

\[
\text{active}(\alpha_C(\kappa), \alpha(i)) = \text{active}(\kappa, i)
\]

\[
\text{pc}(\text{lstate}(\alpha_C(\kappa), \alpha(i))) = \text{pc}(\alpha_S(\text{lstate}(\kappa, i)))
\]

By Definition 6.3.1, we have \(\text{pc}(\alpha_S(\text{lstate}(\kappa, i))) = \text{pc}(\text{lstate}(\kappa, i))\) for every \(i \in 1..N\). Hence, we have \(\text{pc}(\text{lstate}(\alpha_C(\kappa), \alpha(i))) = \text{pc}(\text{lstate}(\kappa, i))\). For every \(i \in 1..N\), we have \(\text{Enabled}(\kappa, i, \text{Loc}_{\text{snd}}) \Leftrightarrow \text{Enabled}(\alpha_C(\kappa), \alpha(i), \text{Loc}_{\text{snd}})\) by the definition of \(\text{Enabled}\) in Section 6.2.2. It implies that a process \(\text{lstate}(\kappa, s)\) is enabled in this sub-round if and only if a process \(\text{lstate}(\alpha_C(\kappa), \alpha(s))\) is enabled in this sub-round for every \(s \in 1..N\).

Now we focus on Constraint (a) by examining processes which are not enabled in this sub-round Send. Let \(i\) be an arbitrary index in \(1..N\) such that \(\neg\text{Enabled}(\kappa, i, \text{Loc}_{\text{snd}})\). It follows that \(\text{lstate}(\kappa, i) \xrightarrow{\text{stutter}} \text{lstate}(\kappa', i)\) by the semantics of the sub-round Send in Section 6.2.2. By Definition 6.3.2, we have

\[
\text{Enabled}(\alpha_C(\kappa), \alpha(i), \text{Loc}_{\text{snd}}) = \text{Enabled}(\kappa, i, \text{Loc}_{\text{snd}})
\]
We show that

\[ Frozen_S(\alpha_C(\kappa), \alpha_C(\kappa'), \alpha(i)) \]

as follows. By Definition 6.3.2, we have

\[ \alpha_S(lstate(\kappa, i)) = lstate(\alpha_C(\kappa), \alpha(i)) \]
\[ \alpha_S(lstate(\kappa', i)) = lstate(\alpha_C(\kappa'), \alpha(i)) \]

By Proposition 6.4.4, it follows that \( \alpha_S(lstate(\kappa, i)) \xrightarrow{\text{stutter}} \alpha_S(lstate(\kappa', i)) \). It follows \( lstate(\alpha_C(\kappa), \alpha(i)) \xrightarrow{\text{stutter}} lstate(\alpha_C(\kappa'), \alpha(i)) \). We now examine the control component for a process \( p_i \). By Definition 6.3.2, we have

\[ active(\kappa, i) = active(\alpha_C(\kappa), \alpha(i)) \]
\[ active(\kappa', i) = active(\alpha_C(\kappa'), \alpha(i)) \]

By definition of \( Frozen_S \) in Section 6.2.2, we have \( active(\kappa, i) = active(\kappa', i) \). It follows \( active(\alpha_C(\kappa), \alpha(i)) = active(\alpha_C(\kappa'), \alpha(i)) \). We now show that each outgoing message buffer from a process \( p_i \) is unchanged from \( \alpha_C(\kappa) \) to \( \alpha_C(\kappa') \). By Definition 6.3.2, we have \( buf(\alpha_C(\kappa'), \alpha(i), \alpha(\ell)) = buf(\kappa', i, \ell) \). Since \( buf(\kappa', i, \ell) = buf(\kappa, i, \ell) \), (we are examining inactive processes in this sub-round Send), it follows \( buf(\alpha_C(\kappa'), \alpha(i), \alpha(\ell)) = buf(\kappa, i, \ell) \). By Definition 6.3.2 we have \( buf(\kappa, i, \ell) = buf(\alpha_C(\kappa), \alpha(i), \alpha(\ell)) \). It follows that

\[ buf(\alpha_C(\kappa'), \alpha(i), \alpha(\ell)) = buf(\alpha_C(\kappa), \alpha(i), \alpha(\ell)) \]

Therefore, it follows \( Frozen_S(\alpha_C(\kappa), \alpha_C(\kappa'), \alpha(i)) \). Constraint (a) holds.

Now we focus on Constraint (b) by examining processes which are enabled in this sub-round Send. Let \( s \in 1..N \) be an arbitrary index such that \( active(\kappa, i) \). By the semantics of the sub-round Send in Section 6.2.2, it follows \( lstate(\kappa, s) \xrightarrow{\text{csnd}(m)} lstate(\kappa', s) \). We have \( lstate(\kappa, s) = lstate(\alpha_C(\kappa), \alpha(s)) \) and \( lstate(\kappa', s) = lstate(\alpha_C(\kappa'), \alpha(s)) \) by Definition 6.3.2. By Proposition 6.4.4, it follows

\[ lstate(\alpha_C(\kappa), \alpha(s)) \xrightarrow{\text{csnd}(m)} lstate(\alpha_C(\kappa'), \alpha(s)) \]

We show that \( m \) is new in buffers \( buf(\alpha_C(\kappa), \alpha(s), 1), \ldots, buf(\alpha_C(\kappa), \alpha(s), 1) \). By Definition 6.3.2, for every \( s, r \in 1..N \), we have

\[ buf(\alpha_C(\kappa), \alpha(s), \alpha(r)) = buf(\kappa, s, r) \]
\[ buf(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = buf(\kappa', s, r) \]

We have \( m \notin buf(\kappa, s, r) \) and \( m \in buf(\kappa', s, r) \) by the semantics of the sub-round Send in Section 6.2.2. It follows

\[ m \notin buf(\alpha_C(\kappa), \alpha(s), \alpha(r)) \quad \text{and} \quad m \in buf(\alpha_C(\kappa), \alpha(s), \alpha(r)) \]
In other words, the message $m$ is new in the buffer $buf(\alpha_C(\kappa), \alpha(s), \alpha(r))$. As a result, Constraint (b) holds.

Now we focus on Constraint (c). Let $s \in 1..N$ be an arbitrary index such that $\text{Enabled}(\kappa, i, \text{Loc}_{snd})$. By arguments at the beginning of the proof of Proposition 6.4.15, we have $\text{Enabled}(\alpha_C(\kappa), \alpha(i), \text{Loc}_{snd})$. By the semantics of the sub-round Send in Section 6.2.2, we have $\neg \text{active}(\kappa', i)$. By Definition 6.3.2, we have $\text{active}(\kappa', i) = \text{active}(\alpha_C(\kappa'), \alpha(i))$. It follows $\neg \text{active}(\alpha_C(\kappa'), \alpha(i))$. Constraint (c) holds.

It implies that $\alpha_C(\kappa) \xrightarrow{\text{snd}} \alpha_C(\kappa')$.

3. Sub-round Receive. By similar arguments in the case of the sub-round Send, we have that Constraints (a) and (c) in the sub-round Receive holds. In the following, we focus on Constraint (b). By similar arguments in the case of the sub-round Send, we have that a process $\text{llstate}(\kappa, s)$ is enabled in this sub-round Receive if and only if a process $\text{llstate}(\alpha_C(\kappa), \alpha(s))$ is enabled in this sub-round Receive for every $s \in 1..N$. Hence, we focus on processes which are enabled in this sub-round Receive.

Let $r \in 1..N$ be an index such that $\text{Enabled}(\kappa, i, \text{Loc}_{rcv})$. By the semantics of the sub-round Receive in Section 6.2.2, we have $\text{llstate}(\kappa, r) \xrightarrow{\text{crcv}(S_1, \ldots, S_N)} \text{llstate}(\kappa', r)$ for some sets $S_1, \ldots, S_N \subseteq \text{Set}(\text{Msg})$ of messages. By Definition 6.3.1, we have

$$\alpha_S(\text{llstate}(\kappa, r)) = \text{llstate}(\alpha_C(\kappa), \alpha(r))$$

and

$$\alpha_S(\text{llstate}(\kappa', r)) = \text{llstate}(\alpha_C(\kappa'), \alpha(r))$$

By Proposition 6.4.3, it follows

$$\text{llstate}(\alpha_C(\kappa), \alpha(r)) \xrightarrow{\alpha_R(\text{crcv}(S_1, \ldots, S_N))} \text{llstate}(\alpha_C(\kappa'), \alpha(r))$$

Now we focus on the update of message buffers. By the semantics of the sub-round Receive in Section 6.2.2, we have $S_s \subseteq buf(\kappa, s, r)$ for every $s \in 1..N$. By Definition 6.3.2, we have $buf(\kappa, s, r) = buf(\alpha_C(\kappa), \alpha(s), \alpha(r))$. It follows that $S_s \subseteq buf(\alpha_C(\kappa), \alpha(s), \alpha(r))$ for every $s \in 1..N$. We now prove that for every $s \in 1..N$, $S_s$ is removed from the message buffer $buf(\alpha_C(\kappa'), \alpha(s), \alpha(r))$. By Definition 6.3.2, we have

$$buf(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = buf(\kappa', s, r)$$

and

$$buf(\alpha_C(\kappa), \alpha(s), \alpha(r)) = buf(\kappa, s, r)$$

By the semantics of the sub-round Receive in Section 6.2.2, we have

$$S_s \cap buf(\kappa', s, r) = \emptyset$$

and

$$buf(\kappa, s, r) = buf(\kappa', s, r) \cup S_s$$

It follows that

$$S_s \cap buf(\alpha_C(\kappa'), \alpha(s), \alpha(r)) = \emptyset$$
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

\[ \text{buf}(\alpha_C(\kappa),\alpha(s),\alpha(r)) = \text{buf}(\alpha_C(\kappa'),\alpha(s),\alpha(r)) \cup S \]

Constraint (b) holds. It implies that \( \alpha_C(\kappa') \xrightarrow{\text{RCV}} \alpha_C(\kappa) \).


Therefore, Proposition 6.4.7 holds.

Proposition 6.4.8 Let \( G_N = (C_N, T_N, R_N, q_0^N) \) be a global transition system with indexes 1..N and a process template \( U_N = (Q_N, Tr_N, Rel_N, q_0_N) \). Let \( \kappa \) and \( \kappa' \) be configurations in \( G_N \) such that \( \kappa \leadsto \kappa' \). Let \( \alpha \) be a transposition on 1..N, and \( \alpha_C \) be a global transposition based on \( \alpha \) (from Definition 6.3.2). It follows \( \alpha_C(\kappa) \leadsto \alpha_C(\kappa') \).

Proof. It immediately follows by Proposition 6.4.7 and the fact that for all \( i \in 1..N \), we have \( \text{active}(\alpha_C(\kappa),\alpha(i)) = \text{active}(\kappa, i) \).

6.4.3 Trace Equivalence of \( G_2 \) and \( G_N \) under \( AP\{1,2\} \)

Recall that \( G_2 \) and \( G_N \) are two global transition systems of 2 and \( N \) processes, respectively, such that every correct process runs the same arbitrary symmetric point-to-point algorithm, and a set \( AP\{1,2\} \) contains predicates that takes one of the forms: \( P_1(1), P_2(2), P_3(1,2), \) or \( P_4(2,1) \) where 1 and 2 are process indexes.

Proposition 6.4.9 Let \( A \) be an arbitrary symmetric point-to-point algorithm. Let \( U_2 \) and \( U_N \) be two process templates of \( A \) for some \( N \geq 2 \), and \( \rho^N \in Q_N \) be a state of \( U_N \). Let \( \rho^2 \) be a tuple that is the application of Construction 6.3.1 to \( \rho^N \) and indexes \( \{1,2\} \). Then, \( \rho^2 \) is a template state of \( U_2 \).

Proof. It immediately follows by Construction 6.3.1.

Proposition 6.4.10 Let \( A \) be an arbitrary symmetric point-to-point algorithm. Let \( G_2 \) and \( G_N \) be two global transition systems of two instances of \( A \) for some \( N \geq 2 \), and \( \kappa_N \in C_N \) be a global configuration in \( G_N \). Let \( \kappa^2 \) be a tuple that is the application of Construction 6.3.2 to \( \kappa^N \) and indexes \( \{1,2\} \). Then, \( \kappa^2 \) is a global configuration of \( G_2 \).

Proof. It immediately follows by Construction 6.3.2.

Lemma 6.3.3 Let \( A \) be a symmetric point-to-point algorithm. Let \( G_2 \) and \( G_N \) be two transition systems such that all processes in \( G_2 \) and \( G_N \) follow \( A \), and \( N \geq 2 \). Let \( \pi^N = \kappa_0^N \kappa_1^N \ldots \) be an admissible sequence of configurations in \( G_N \). Let \( \pi^2 = \kappa_0^2 \kappa_1^2 \ldots \) be a sequence of configurations in \( G_2 \) such that \( \kappa_k^2 \) is the index projection of \( \kappa_k^N \) on indexes \( \{1,2\} \) for every \( k \geq 0 \). Then, \( \pi^2 \) is admissible in \( G_2 \).

The proof of Lemma 6.3.3 requires the following propositions:
6.4. Detailed Proofs for Cutoff Results in the Model of the Global Distributed Systems

1. Proposition [6.4.11] says that the application of Construction [6.3.1] to an initial template state of $G_N$ constructs an initial template state of $G_2$.


Proof of Lemma 6.3.3. It is easy to check that Lemma 6.3.3 holds by Propositions 6.4.11, 6.4.13, 6.4.14, 6.4.15, and 6.4.16. The detailed proofs of these propositions are given below.

Proposition 6.4.11 Let $\mathcal{A}$ be an arbitrary symmetric point-to-point algorithm. Let $U_N = (Q_N, T_{\mathcal{R}N}, \text{Rel}_N, q^0_N)$, $U_2 = (Q_2, T_{\mathcal{R}2}, \text{Rel}_2, q^0_2)$ be two process templates of $\mathcal{A}$ for some $N \geq 2$. It follows that $q^0_2$ is the index projection of $q^0_N$ on indexes $\{1, 2\}$.

Proof. It follows by Construction 6.3.1 and the definition of $q^0_2$ in Section 6.2.1.

Proposition 6.4.12 Let $\mathcal{A}$ be an arbitrary symmetric point-to-point algorithm, and $U_2$ and $U_N$ be process templates of $\mathcal{A}$. Let $\rho_0$ and $\rho_1$ be template states in $U_N$ such that $\rho_0 \xrightarrow{\text{crcv}(S_1, \ldots, S_N)} \rho_1$ for some sets $S_1, \ldots, S_N$ of messages. Let $\rho'_0$ and $\rho'_1$ be template states of $U_2$ such that they are constructed with Construction 6.3.2 and based on configurations $\rho_0$ and $\rho_1$ and indexes $\{1, 2\}$. It follows $\rho'_0 \xrightarrow{\text{rcv}(S_1, S_2)} \rho'_1$.

Proof. We prove that all Constraints (a)–(c) for the transition $\text{csnd}$ defined in Section 6.2.1 hold. First, we focus on Constraint (a). We have $pc(\alpha_S(\rho_1)) = pc(\rho_1)$ by Definition 6.3.1. We have $pc(\rho_1) = \text{nextLoc}(pc(\rho_0))$ by the semantics of $\text{crcv}(S_1, \ldots, S_N)$ in Section 6.2.1. We have $\text{nextLoc}(pc(\rho_0)) = \text{nextLoc}(pc(\rho'_0))$ by Definition 6.3.1. Hence, it follows $pc(\alpha_S(\rho_1)) = \text{nextLoc}(pc(\alpha_S(\rho_0)))$. Moreover, we have $\emptyset = \text{genMsg}(pc(\rho_0))$ by the semantics of $\text{crcv}$ in Section 6.2.1. By Construction 6.3.1, we have $\text{genMsg}(pc(\rho_0)) = \text{genMsg}(pc(\rho'_0))$. It follows that $\text{genMsg}(pc(\rho'_0)) = \emptyset$. Constraint (a) holds.

We now check components related to received messages (Constraint (b)). Let $i$ be an arbitrary index in $1..2$. It follows

$$\text{rcvd}(\rho'_1, i)$$
$$= \text{rcvd}(\rho_1, i) \quad \text{(by Construction 6.3.1)}$$
$$= \text{rcvd}(\rho_0, i) \cup S_i \quad \text{(by the semantics of $\text{crcv}(S_1, \ldots, S_N)$ in Section 6.2.1)}$$
$$= \text{rcvd}(\rho'_0, i) \cup S_i \quad \text{(by Construction 6.3.1)}$$
Hence, we have \( \forall i \in 1..2: rcvd(\rho'_1, i) = rcvd(\rho'_0, i) \cup S_i \). Constraint (b) holds. Moreover, by similar arguments in the proof of Proposition 6.4.13, we have

\[
\forall i \in 1..2: lvar(\rho'_1, i) = nextVar(pc(\rho'_0), S_i, lvar(\rho'_0, i))
\]

Constraint (c) holds. It follows \( \rho'_0 \xrightarrow{rcv(S_1, S_2)} \rho'_1 \).

**Proposition 6.4.13** Let \( \mathcal{A} \) be an arbitrary symmetric point-to-point algorithm, and \( \mathcal{U}_2 \) and \( \mathcal{U}_N \) be process templates of \( \mathcal{A} \). Let \( \tau r \in \mathcal{T}_N \) be a transition such that it is one of send, computation, crash or stuttering transitions. Let \( \rho_0 \) and \( \rho_1 \) be the template states in \( \mathcal{U}_N \) such that \( \rho_0 \xrightarrow{\tau r} \rho_1 \). Let \( \rho'_0 \) and \( \rho'_1 \) be states of \( \mathcal{U}_2 \) such that they are the index projection of \( \rho_0 \) and \( \rho_1 \) on indexes \( \{1,2\} \), respectively. Then, \( \rho'_0 \xrightarrow{\tau r} \rho'_1 \).

**Proof.** By similar arguments in the proof of Lemma 6.4.12.

Now we turn to properties of the global configurations under Constructions 6.3.2.

**Proposition 6.4.14** Let \( \mathcal{A} \) be an arbitrary symmetric point-to-point algorithm. Let \( G_N = (C_N, T_N, R_N, g^0_N) \) and \( G_2 = (C_2, T_2, R_2, g^0_2) \) be two transition systems of two instances of \( \mathcal{A} \) for some \( N \geq 2 \). It follows that \( g^0_2 \) is the index projection of \( g^0_N \) on indexes \( \{1,2\} \).

**Proof.** It immediately follows by Construction 6.3.2 and the definition of \( g^0_N \).

**Proposition 6.4.15** Let \( \mathcal{A} \) be an arbitrary symmetric point-to-point algorithm. Let \( G_2 = (C_2, T_2, R_2, g^0_2) \) and \( G_N = (C_N, T_N, R_N, g^0_N) \) be global transition systems such that all processes in \( G_2 \) and \( G_N \) follow the same algorithm \( \mathcal{A} \) for some \( N \geq 2 \). Let \( \kappa_0 \) and \( \kappa_1 \) be global configurations in \( G_N \) such that \( \kappa_0 \xrightarrow{\tau r} \kappa_1 \) where \( \tau r \) is an internal transition. Let \( \kappa'_0 \) and \( \kappa'_1 \) be the index projection of \( \kappa_0 \) and \( \kappa_1 \) on a set \( \{1,2\} \) of indexes, respectively. It follows that \( \kappa'_0 \xrightarrow{\tau r} \kappa'_1 \).

**Proof.** First, by Proposition 6.4.10, both \( \rho'_0 \) and \( \rho'_1 \) are configurations in \( G_2 \). We prove Proposition 6.4.15 by case distinction. Here we provide detailed proofs only of two sub-rounds Schedule and Send. The proofs of other sub-rounds are similar.

- **Sub-round Schedule.** We prove that all Constraints (a)-(c) hold in \( \kappa'_0 \) and \( \kappa'_1 \). We now focus on Constraint (a). We have \( active(\kappa'_0, 1) = active(\kappa_0, 1) \) and \( active(\kappa_0, 2) = active(\kappa_0, 2) \) by Construction 6.3.2. By the semantics of the sub-round Schedule in Section 6.2.2, we have \( \neg active(\kappa_0, 1) \land \neg active(\kappa_0, 2) \). Hence, it follows \( \neg active(\kappa'_0, 1) \land \neg active(\kappa'_0, 2) \). Hence, the sub-round Schedule can start with a configuration \( \kappa'_0 \). Constraint (a) holds. By Proposition 6.4.13, Constraint (b) holds. Constraint (c) holds by Construction 6.3.2. Now we focus on incoming message buffers to correct processes to prove Constraint (d). Let \( r \) be an index in \( 1..N \)
6.4. Detailed Proofs for Cutoff Results in the Model of the Global Distributed Systems

such that $pc(lstate(\kappa'_1, r)) \neq \ell_{\text{crash}}$. By Construction 6.3.2, for every $s \in 1..2$, we have $buf(\kappa'_1, s, r) = buf(\kappa_1, s, r)$ and $buf(\kappa_0, s, r) = buf(\kappa'_0, s, r)$. By the semantics of the sub-round Schedule in Section 6.2.2 we have $buf(\kappa_1, s, r) = buf(\kappa_0, s, r)$ for every $s \in 1..2$. It follows $buf(\kappa'_1, s, r) = buf(\kappa'_0, s, r)$ for every $s \in 1..2$. Constraint (d) holds. Now we focus on message buffers to crashed processes to prove Constraint (d). Let $r$ be an index in $1..N$ such that $pc(lstate(\kappa'_1, r)) = \ell_{\text{crash}}$. By Construction 6.3.2, for every $s \in 1..2$, we have $buf(\kappa'_1, s, r) = buf(\kappa_1, s, r)$. By the semantics of the sub-round Schedule in Section 6.2.2, we have $buf(\kappa_1, s, r) = \emptyset$ for every $s \in 1..2$. It follows $buf(\kappa'_1, s, r) = \emptyset$ for every $s \in 1..2$. Constraint (e) holds. It implies that $\kappa'_0 \xrightarrow{\text{sched}} \kappa'_1$.

- **Sub-round Send.** For every $k \in 1..2$, we have $active(\kappa'_0, 1) = active(\kappa_0, 1)$ and $active(\kappa'_0, 2) = active(\kappa_0, 2)$. Hence, if a process $p^N_i$ in $G_N$ is enabled in this sub-round, a corresponding process $p^2_i$ in $G_N$ is also for every $i \in 1..2$. We prove that all Constraints (a)--(c) between $\kappa'_0$ and $\kappa'_1$ for the sub-round Send defined in Section 6.2.2 hold. By similar arguments in the proof of Proposition 6.4.15 Constraint (a) holds. Now we focus on enable processes to prove Constraints (b) and (c).

Assume that a process $p^N_i$ in $G_N$ has sent a message $m$ in this sub-round, we show that a process $p^2_i$ in $G_2$ has also sent the message $m$ in this sub-round where $i \in 1..2$.

By Proposition 6.4.13, it follows that $lstate(\kappa'_0, i) \xrightarrow{\text{csnd}(m)} lstate(\kappa'_1, i)$. Now we show that the message $m$ is new in buffers $buf(\kappa'_1, i, 1)$ and $buf(\kappa'_1, i, 2)$. By Construction 6.3.2, we have $buf(\kappa'_1, i, \ell) = buf(\kappa_1, i, \ell)$ and $buf(\kappa'_0, i, \ell) = buf(\kappa_0, i, \ell)$ for every $\ell \in 1..2$. By the semantics of the sub-round Send in Section 6.2.2, we have $buf(\kappa_1, i, \ell) = \{m\} \cup buf(\kappa_0, i, \ell)$. It follows $buf(\kappa'_1, i, \ell) = \{m\} \cup buf(\kappa'_0, i, \ell)$ for every $\ell \in 1..2$. Moreover, since $m \notin buf(\kappa_0, i, \ell)$ for every $\ell \in 1..2$, we have $m \notin buf(\kappa'_0, i, \ell)$. Therefore, the message $m$ is new in a buffer $buf(\kappa'_1, i, \ell)$ for every $\ell \in 1..2$. Constraint (b) holds. Moreover, by Construction 6.3.2 we have $active(\kappa_1, i) = active(\kappa'_1, i)$. We have $\neg active(\kappa'_1, i)$ by the semantics of the sub-round Send in Section 6.2.2. It follows $\neg active(\kappa'_1, i)$. Constraint (e) holds. It follows that $\kappa'_0 \xrightarrow{\text{csnd}} \kappa'_1$.

- **Sub-rounds Receive and Computation.** Similar.

Therefore, Proposition 6.4.15 holds.

**Proposition 6.4.16** Let $A$ be an arbitrary symmetric point-to-point algorithm. Let $G_2$ and $G_N$ be global transition systems of $A$ for some $N \geq 2$. Let $\kappa_0$ and $\kappa_1$ be global configurations of $C_N$ such that $\kappa_0 \sim \kappa_1$. Let $\kappa'_0$ and $\kappa'_1$ be the index projection of $\kappa_0$ and $\kappa_1$ on indexes $\{1, 2\}$. Then, $\kappa'_0 \sim \kappa'_1$.

**Proof.** It immediately follows by Propositions 6.4.13 and 6.4.15.
Now we present how to construct an admissible path of $G_N$ from a given admissible path of $G_2$ with Lemma 6.3.4 below. The main argument is that from an admissible sequence of configurations in $G_2$, we can get an admissible sequence of configurations in $G_N$ by letting processes 3 to $N$ be initially crashed. The proof of Lemma 6.3.4 requires the preliminary Propositions 6.4.17 and 6.4.18.

**Proposition 6.4.17** Let $A$ be an arbitrary symmetric point-to-point algorithm. Let $G_2$ and $G_N$ be global transition systems of $A$ for some $N \geq 2$. Let $\kappa^2$ be a configuration in $G_2$. There exists a configuration $\kappa^N$ in $G_N$ such that the following properties hold:

- $\kappa^2$ is the index projection of $\kappa^N$ on indexes $\{1, 2\}$, and
- $\forall i \in 3..N: pc(lstate(\kappa^N, i)) = \ell_{\text{crash}} \land \neg\text{active}(\kappa^N, i)$
- $\forall s \in 3..N, r \in 1..N: \text{buf}(\kappa, s, r) = \emptyset$
- $\forall s \in 3..N, r \in 1..2: \text{rcvd}(lstate(\kappa^N, s), r) = \emptyset$
- $\forall s \in 1..N, r \in 3..N: \text{buf}(\kappa, s, r) = \emptyset$

**Proof.** Proposition 6.4.17 is true since our construction adds $N-2$ crashed processes that have not sent any messages in the global system. The last two constraints requires that processes 1 and 2 have not received any messages from crashed processes and the message buffers to crashed processes are empty. Other components are arbitrary. ■

**Proposition 6.4.18** Let $A$ be an arbitrary symmetric point-to-point algorithm. Let $G_2$ and $G_N$ be global transition systems of $A$ for some $N \geq 2$. Let $\kappa_0^2$ and $\kappa_1^2$ be configurations in $G_2$ such that $\kappa_0^2 \xrightarrow{\text{tr}} \kappa_1^2$ where $\text{tr}$ is an internal transition. There exists two configurations $\kappa_0^N$ and $\kappa_1^N$ in $G_N$ such that

1. $\kappa_0^2$ and $\kappa_1^2$ are respectively the index projection of $\kappa_0^N$ and $\kappa_1^N$ on a set $\{1, 2\}$ of indexes, and
2. $\kappa_0^N \xrightarrow{\text{tr}} \kappa_1^N$.

**Proof.** By Proposition 6.4.17, there exists a configuration $\kappa_0^N$ in $G_N$ such that (i) $\kappa_0^2$ is the index projection of $\kappa_0^N$ on indexes $\{1, 2\}$, and (ii) all processes with indexes in $3..N$ are crashed and inactive in $\kappa_0^N$, and (iii) every process $p_i$ has not received any messages from a process $p_j$ where $i \in \{1, 2\}, j \in 3..N$ (as the configuration construction in the proof of Proposition 6.4.17).

We construct $\kappa_1^N$ as the following. Intuitively, this construction keeps process $3..N$ crashed, and two processes 1 and 2 in $G_N$ make similar transitions with processes 1 and 2 in $G_2$. 
1. \( \forall i \in 3..N : \text{lstate}(\kappa_1^N, i) = \text{lstate}(\kappa_0^N, i) \land \neg \text{active}(\kappa_1^N, i) \).
2. \( \forall s \in 1..N, r \in 3..N : \text{buf}(\kappa_1^N, s, r) = \emptyset \)
3. \( \forall i \in \{1, 2\} : \text{active}(\kappa_1^N, i) = \text{active}(\kappa_0^N, i) \)
4. \( \forall s \in 3..N, r \in 1..N : \text{rcvd}(\text{lstate}(\kappa_1^N, s), r) = \emptyset \)
5. \( \forall s, r \in \{1, 2\} : \text{buf}(\kappa_1^N, s, r) = \text{buf}(\kappa_1^N, s, r) \)
6. For every \( s \in \{1, 2\} \), we have: If \( \text{lstate}(\kappa_0^2, s) \xrightarrow{\text{csnd}(m)} \text{lstate}(\kappa_1^2, s) \) for some \( m \in \text{Ms} \), then \( \text{buf}(\kappa_1^N, s, r) = \{m\} \cup \text{buf}(\kappa_0^N, s, r) \) for every \( 3 \leq r \leq N \). Otherwise, \( \text{buf}(\kappa_1^N, s, r) = \text{buf}(\kappa_0^N, s, r) \).

7. For every \( i \in \{1, 2\} \), the configurations of processes with index \( i \) is updated as following:
   - \( \text{pc}(\text{lstate}(\kappa_1^N, i)) = \text{pc}(\text{lstate}(\kappa_1^2, i)) \)
   - \( \forall j \in \{1, 2\} : \text{rcvd}(\text{lstate}(\kappa_1^N, i), j) = \text{rcvd}(\text{lstate}(\kappa_1^2, i), j) \)
   - \( \forall j \in \{1, 2\} : \text{lvar}(\text{lstate}(\kappa_1^N, i), j) = \text{lvar}(\text{lstate}(\kappa_1^2, i), j) \)
   - \( \forall j \in 3..N : \text{rcvd}(\text{lstate}(\kappa_1^N, i), j) = \text{rcvd}(\text{lstate}(\kappa_0^N, i), j) = \emptyset \)
   - \( \forall j \in 3..N : \text{lvar}(\text{lstate}(\kappa_1^N, i), j) = \text{nextVar}(\text{pc}(\text{lstate}(\kappa_0^0, i)), \emptyset, \text{lvar}(\text{lstate}(\kappa_0^0, i), j)) \)

By the construction of \( \kappa_1^N \), it follows that \( \kappa_1^N \) is a configuration in \( \mathcal{G}_N \). Moreover, we have \( \text{lstate}(\kappa_0^0, i) \xrightarrow{\text{stutter}} \text{lstate}(\kappa_1^N, i) \) and \( \text{lstate}(\kappa_0^0, i) \) is crashed for every \( i \in 3..N \). Moreover, no message from a process \( p_i \) has been sent or received for every \( i \in 3..N \).

By the above construction, it immediately follows that \( \kappa_1^2 \) is the index projection \( \kappa_1^N \) on indexes \( \{1, 2\} \). Hence, point 1 in Proposition 6.4.18 holds. We prove point 2 in Proposition 6.4.18 by case distinction.

- **Sub-round Schedule.** By similar arguments in the proof of Proposition 6.4.15 and the construction of \( \kappa_1^N \), we have \( \kappa_1^N \xrightarrow{\text{stutter}} c_2^N \).

- **Sub-round Send.** By construction of \( \kappa_0^N \) and \( \kappa_1^N \), we know that every process \( p_i \) is crashed, and its state is not updated, and every outgoing message buffer from \( p_i \) is always empty for every \( i \in 3..N \). Therefore, in the following, we focus on only two processes \( p_1 \) and \( p_2 \). For every \( i \in 1..2 \), if \( \neg \text{Enabled}(c_0^N, c_1^N, \text{Loc}_{\text{snd}}) \), it follows \( \text{Frozen}_S(\kappa_0^N, \kappa_1^N, i) \) by the construction of configurations \( \kappa_0^N \) and \( \kappa_1^N \). Constraint (a) holds. We now focus on enabled processes in this sub-round. Let \( i \) be an index in 1.2 such that \( \text{Enabled}(c_0^N, c_1^N, \text{Loc}_{\text{snd}}) \). By the semantics of the sub-round Send in Section 6.2.1, we have \( \text{lstate}(\kappa_0, i) \xrightarrow{\text{csnd}(m)} \text{lstate}(\kappa_1, i) \). We prove that \( \text{lstate}(\kappa_0^N, i) \xrightarrow{\text{csnd}(m)} \text{lstate}(\kappa_1^N, i) \) as follows. By the construction of \( \kappa_0^N \), we have that \( \text{pc}(\text{lstate}(\kappa_0^0, i)) = \text{pc}(\text{lstate}(\kappa_0, i)) \). By the construction of \( \kappa_1^N \), we have that
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

\[ pc(lstate(κ_1^N, i)) = pc(lstate(κ_1, i)) \]

By the semantics of the transition \( csnd(m) \) in Section 6.2.1, we have \( pc(lstate(κ_1, i)) = nextLoc(pc(lstate(κ_0, i))) \). It follows that

\[ pc(lstate(κ_0^N, i)) = nextLoc(pc(lstate(κ_0^N, i))) \]

By similar arguments, it follows \( \{ m \} = genMsg(pc(lstate(κ_0^N, i))) \). Now we focus on received messages of a process \( p_i \). By the semantics of the transition \( csnd(m) \) in Section 6.2.1, we have \( rcvd(lstate(κ_2^N, i), j) = rcvd(lstate(κ_0^N, i), j) \). For every \( j \in \{1, 2\} \), we have

\[
\begin{align*}
rcvd(lstate(κ_1^N, i), j) &= rcvd(lstate(κ_1, i), j) \quad \text{(by the construction of } κ_1^N) \\
&= rcvd(lstate(κ_0, i), j) \quad \text{(by the semantics of } csnd(m) \text{ in Section 6.2.1)} \\
&= rcvd(lstate(κ_0^N, i), j) \quad \text{(by the construction of } κ_0^N)
\end{align*}
\]

Hence, a process \( p_i \) does not receive any message when taking a step from \( κ_0^N \) to \( κ_1^N \). By the construction of \( κ_1^N \), it follows

\[
\begin{align*}
lvar(lstate(κ_1^N, i), j) &= \emptyset \quad \text{(by the construction of } κ_1^N) \\
&= lvar(lstate(κ_0^N, i), j) \quad \text{(by the construction of } κ_0^N)
\end{align*}
\]

for every \( j \in \{1, N\} \). Hence, it follows \( lstate(κ_0^N, i) \xrightarrow{csnd(m)} lstate(κ_1^N, i) \). Now we focus on outgoing message buffers from \( p_i \). It is easy to see that the message \( m \) is new in every message buffer \( buf(κ_2^N, i, j) \). By construction of \( κ_0^N \) and \( κ_1^N \), it follows that \( m \notin buf(κ_0^N, i, j) \) and \( m \in buf(κ_1^N, i, j) \). By similar arguments, we have \( buf(κ_0^N, i, j) = \{ m \} \cup buf(κ_1^N, i, j) \). Hence, Constraint (b) between \( κ_0^N \) and \( κ_1^N \) in the sub-round Send in Section 6.2.2 hold. Moreover, by the construction of \( κ_1^N \), we have \( active(κ_1^N, i) = active(κ_1, i) \). By the semantics of the sub-round Send in Section 6.2.2, we have \( \neg active(κ_1^N, i) \). It follows \( \neg active(κ_1^N, i) \). Constraint (c) between \( κ_0^N \) and \( κ_1^N \) in the sub-round Send in Section 6.2.2 holds. Therefore, it follows \( κ_0^N \xrightarrow{\text{comp}} κ_1^N \).

- **Case** \( κ_0 \xrightarrow{\text{rcv}} κ_1 \). It follows by similar arguments of the sub-round Send, except that if \( lstate(κ_0, i) \xrightarrow{rcv(S_1, S_2)} lstate(κ_1, i) \), then

\[
lstate(κ_0^N, i) \xrightarrow{rcv(S_1, S_2, \emptyset, \emptyset)} lstate(κ_1^N, i)
\]

- **Case** \( κ_0 \xrightarrow{\text{comp}} κ_1 \). By similar arguments in the case of the sub-round Send.
6.4. Detailed Proofs for Cutoff Results in the Model of the Global Distributed Systems

Then, Proposition 6.4.18 holds. □

Notice that in Proposition 6.4.18, if both \( p_1^N \) and \( p_2^N \) take a stuttering step from \( \kappa_0^N \) to \( \kappa_1^N \), then \( \kappa_0^N = \kappa_1^N \).

**Lemma 6.3.4.** Let \( \mathcal{A} \) be an arbitrary symmetric point–to–point algorithm. Let \( \mathcal{G}_N \) be global transition systems of \( \mathcal{A} \) for some \( N \geq 2 \). Let \( \pi^2 = \kappa_0^N \kappa_1^2 \ldots \) be an admissible sequence of configurations in \( \mathcal{G}_2 \). There exists an admissible sequence \( \pi^N = \kappa_0^N \kappa_1^N \ldots \) of configurations in \( \mathcal{G}_N \) such that \( \kappa_i^2 \) is the index projection of \( \kappa_i^N \) on indexes \( \{1, 2\} \) for every \( i \geq 0 \).

**Proof.** We prove Lemma 6.3.4 by inductively constructing \( \kappa_k^N \).

**Base case.** Since \( \pi^2 \) and \( \pi^N \) are admissible sequences, it follows \( \kappa_0^2 = g_0^0 \) and \( \kappa_0^N = g_0^N \).

By Proposition 6.4.14, we have that \( \kappa_0^N \) is the index projection of \( \kappa_0^N \) on indexes \( \{1, 2\} \). We construct \( \kappa_1^N \) by scheduling that all processes \( 3..N \) crash in \( \kappa_1^N \). Formally, we have:

1. \( \forall i \in 3..N: \text{pc}(\text{lstate}(\kappa_1^N, i)) = \ell_{\text{crash}} \land \neg \text{active}(\kappa_1^N, i) \).
2. \( \forall i \in \{1, 2\}: \text{active}(\kappa_1^N, i) = \text{active}(\kappa_1^2, i) \land \text{pc}(\text{lstate}(\kappa_1^N, i)) = \text{pc}(\text{lstate}(\kappa_1^2, i)) \).
3. \( \forall s, r \in 1..N: \text{buf}(\kappa_1^N, s, r) = \text{buf}(\kappa_0^N, s, r) \).
4. \( \forall s, r \in 1..N: \text{rcvd}(\text{lstate}(\kappa_1^N, s), r) = \text{rcvd}(\text{lstate}(\kappa_0^N, s), r) \).
5. \( \forall s, r \in 1..N: \text{lvar}(\text{lstate}(\kappa_1^N, s), r) = \text{lvar}(\text{lstate}(\kappa_0^N, s), r) \).

The above constraints ensure that \( \kappa_0^N \xrightarrow{\text{Sched}} \kappa_1^N \).

**Induction step.** It immediately follows by Proposition 6.4.18.

Hence, Lemma 6.3.4 holds. □

**Lemma 6.3.5.** Let \( \mathcal{A} \) be a symmetric point–to–point algorithm. Let \( \mathcal{G}_2 \) and \( \mathcal{G}_N \) be its instances for some \( N \geq 2 \). Let \( AP_{\{1, 2\}} \) be a set of predicates that take one of the forms: \( P_1(1), P_2(2), P_3(1, 2) \) or \( P_4(2, 1) \). It follows that \( \mathcal{G}_2 \) and \( \mathcal{G}_N \) are trace equivalent under \( AP_{\{1, 2\}} \).

**Proof.** It immediately follows by Definition 6.3.4, Lemma 6.3.3 and Lemma 6.3.4. □

6.4.4 Trace Equivalence of \( \mathcal{G}_1 \) and \( \mathcal{G}_N \) under \( AP_{\{1\}} \)

Lemma 6.4.1 says that two global transition systems \( \mathcal{G}_1 \) and \( \mathcal{G}_N \) whose processes follow an arbitrary symmetric point–to–point algorithm are trace equivalent under a set \( AP_{\{1\}} \) of predicates which inspect only variables whose index is 1. The proof of Lemma 6.4.1 is similar to one of Lemma 6.4.1, but applies Constructions 6.4.1 and 6.4.2. Constructions 6.4.1 and 6.4.2 are respectively similar to Constructions 6.3.1 and 6.3.2 but focus on only an index 1. Constructions 6.4.1 and 6.4.2 are used in the proof of Lemma 6.4.1.
Construction 6.4.1 Let $A$ be an arbitrary symmetric point–to–point algorithm. Let $U_N$ be a process template of $A$ for some $N \geq 2$, and $\rho^N$ be a template state of $U_N$. We construct a tuple $\rho^1 = (pc_1, rcvd_1, v_1)$ based on $\rho^N$ and a set $\{1\}$ of process indexes in the following way: $pc_1 = pc(\rho^N)$, $rcvd_1 = rcvd(\rho^N, 1)$, and $v_1 = lvar(\rho^N, 1)$.

Construction 6.4.2 Let $A$ be a symmetric point–to–point algorithm. Let $G_1$ and $G_N$ be two global transitions of two instances of $A$ for some $N \geq 1$, and $\kappa^N \in C_N$ be a global configuration in $G_N$. We construct a tuple $\kappa^2 = (s_1, buf_1, act_1)$ based on $\kappa^N$ and a set $\{1\}$ in the following way: $s_1$ is constructed from $lstate(\kappa^N, 1)$ with Construction 6.4.1 and an index 1, and $buf_1 = buf(\kappa^N, 1, 1)$, and $act_1 = active(\kappa^N, 1)$.

Lemma 6.4.1 Let $A$ be a symmetric point–to–point algorithm. Let $G_1$ and $G_N$ be instances for some $N \geq 2$. Let $AP_{\{1\}}$ be a set of predicates which inspect only variables whose index is 1. It follows that $G_1$ and $G_N$ are trace equivalent under $AP_{\{1\}}$.

Proof. By applying similar arguments in the proof of Lemma 6.3.5 with Constructions 6.4.1 and 6.4.2.

6.4.5 Cutoff results in the unrestricted model
In the following, we prove Propositions 6.4.19 and 6.4.20 which allows us to change positions of big conjunctions in specific formulas. Propositions 6.4.19 and 6.4.20 are used in the proof of our cutoff results, Theorems 6.3.1 and 6.3.2, respectively.

Proposition 6.4.19 Let $A$ be a symmetric point–to–point algorithm. Let $G_N$ be instances of $N$ processes for some $N \geq 1$. Let $Path_N$ be sets of all admissible sequences of configurations in $G_N$. Let $\omega_{\{i\}}$ be a LTL$\backslash$X formula in which every predicate takes one of the forms: $P_1(i)$ or $P_2(i, i)$ where $i$ is an index in $1..N$. Then,

\[
\left( \forall \pi_N \in Path_N: G_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_{\{i\}} \right) \tag{6.1}
\]

\[
\Leftrightarrow \left( \bigwedge_{i \in 1..N} \left( \forall \pi_N \in Path_N: G_N, \pi_N \models \omega_{\{i\}} \right) \right) \tag{6.2}
\]

Proof. ($\Rightarrow$) Let $\pi_N$ be an arbitrary admissible sequence of configurations in $Path_N$ such that $G_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_{\{i\}}$. Let $i_0$ be an arbitrary index in $1..N$, we have $G_N, \pi_N \models \omega_{\{i_0\}}$. Hence, for every $\pi_N \in Path_N$, for every $i_0 \in 1..N$, it follows $G_N, \pi_N \models \omega_{\{i_0\}}$. Therefore, Formula 6.1 implies: for every $i_0 \in 1..N$, for every $\pi_N \in Path_N$, it follows $G_N, \pi_N \models \omega_{\{i_0\}}$. It follows that: for every $i_0 \in 1..N$, $\forall \pi_N \in Path_N: G_N, \pi_N \models \omega_{\{i_0\}}$. Now, we have that Formula 6.1 implies Formula 6.2.

($\Leftarrow$) By applying similar arguments.

Proposition 6.4.20 Let $A$ be a symmetric point–to–point algorithm. Let $G_N$ be instances of $N$ processes respectively for some $N \geq 1$. Let $Path_N$ be sets of all admissible sequences
of configurations in $G_N$. Let $\omega_{(i)}$ be a LTL\$X formula in which every predicate takes one of the forms: $P_1(i)$ or $P_2(i, i)$ where $i$ is an index in $1..N$. Then,

$$\left( \forall \pi_N \in Path_N: G_N, \pi_N \models \bigwedge_{i,j \in 1..N} \psi_{(i,j)} \right) \quad (6.3)$$

$$\iff \left( \bigwedge_{i,j \in 1..N} \left( \forall \pi_N \in Path_N: G_N, \pi_N \models \psi_{(i,j)} \right) \right) \quad (6.4)$$

Proof. $\Rightarrow$ Let $\pi_N$ be an arbitrary admissible sequence of configurations in $Path_N$ such that $G_N, \pi_N \models \bigwedge_{i,j \in 1..N} \psi_{(i,j)}$. Let $i_0$ and $j_0$ be arbitrary indexes in $1..N$ such that $i_0 \neq j_0$, we have that $G_N, \pi_N \models \psi_{(i_0,j_0)}$. Hence, for every $\pi_N \in Path_N$, for every $i_0 \in 1..N$, for every $j_0 \in 1..N$ such that $i_0 \neq j_0$, it follows that $G_N, \pi_N \models \psi_{(i_0,j_0)}$. Therefore, Formula 6.3 implies: for every $i_0 \in 1..N$, for every $j_0 \in 1..N$ such that $i_0 \neq j_0$, for every $\pi_N \in Path_N$, it follows $G_N, \pi_N \models \psi_{(i_0,j_0)}$. It follows that: for every $i_0 \in 1..N$, for every $j_0 \in 1..N$ such that $i_0 \neq j_0$, it holds $\forall \pi_N \in Path_N: G_N, \pi_N \models \psi_{(i_0,j_0)}$. Therefore, Formula 6.3 implies Formula 6.4.

$(\Leftarrow)$ By applying similar arguments. \hfill $\blacksquare$

**Theorem 6.3.1.** Let $A$ be a symmetric point–to–point algorithm under the unrestricted model. Let $G_1$ and $G_N$ be instances of $1$ and $N$ processes respectively for some $N \geq 1$. Let $Path_1$ and $Path_N$ be sets of all admissible sequences of configurations in $G_1$ and in $G_N$, respectively. Let $\omega_{(i)}$ be a LTL\$X formula in which every predicate takes one of the forms: $P_1(i)$ or $P_2(i, i)$ where $i$ is an index in $1..N$. Then, it follows that:

$$\left( \forall \pi_N \in Path_N: G_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_{(i)} \right) \iff \left( \forall \pi_1 \in Path_1: G_1, \pi_1 \models \omega_{(1)} \right)$$

Proof. By Proposition 6.4.19 we have

$$\left( \forall \pi_N \in Path_N: G_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_{(i)} \right) \iff \left( \bigwedge_{i \in 1..N} \left( \forall \pi_N \in Path_N: G_N, \pi_N \models \omega_{(i)} \right) \right)$$

Let $i$ be an index in a set $1..N$. Hence, $\alpha = (i \leftrightarrow 1)$ is a transposition on $1..N$ (*). By Lemma 6.3.2 we have: (i) $\psi_{(\alpha(i))} = \psi_{(1)}$, and (ii) $\alpha(G_N) = G_N$, and (iii) $\alpha(g_N^0) = g_N^0$.

Since $\omega_{(i)}$ is an LTL\$X$ formula, $A \omega_{(i)}$ is a LTL\$X$ formula where $A$ is a path operator in LTL\$X$ (see [CJGK+18]). By the semantics of the operator $A$, it follows that $\forall \pi_N \in Path_N: G_N, \pi_N \models \omega_{(i)}$ if and only if $G_N, g_N^0 \models A \omega_{(1)}$. By point (*), it follows $G_N, g_N^0 \models A \omega_{(1)}$ if and only if $G_N, g_N^0 \models A \omega_{(i)}$. Since an index $i$ is arbitrary, we have $G_N, g_N^0 \models A \omega_{(i)}$ if and only if $G_N, g_N^0 \models A \omega_{(1)}$.

We have that $G_N, g_N^0 \models A \omega_{(i)}$ if and only if $\forall \pi_N \in Path_N: G_N, \pi_N \models \omega_{(i)}$ by the semantics of the operator $A$. It follows $\forall \pi_N \in Path_N: G_N, \pi_N \models \omega_{(i)}$ if and only if $\forall \pi_2 \in Path_2: G_2, \pi_2 \models \omega_{(1)}$ by Lemma 6.4.1. Then, Theorem 6.3.1 holds. \hfill $\blacksquare$
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

**Theorem 6.3.2** Let $A$ be a symmetric point-to-point algorithm under the unrestricted model. Let $G_2$ and $G_N$ be instances of $A$ for some $N \geq 2$. Let $\psi_{\{i,j\}}$ be an LTL formula in which every predicate takes one of the forms: $P_1(i)$, or $P_2(j)$, or $P_3(i,j)$, or $P_4(j,i)$ where $i$ and $j$ are different indexes in $1..N$. It follows that:

$$(\forall \pi_N \in Path_N : G_N, \pi_N \models \bigwedge_{i \neq j} \psi_{\{i,j\}}) \iff (\forall \pi_2 \in Path_2 : G_2, \pi_2 \models \psi_{\{1,2\}})$$

**Proof.** By similar arguments in the proof of Theorem 6.3.1

6.4.6 Verification of the Failure Detector of [CT96] with the Cutoffs

In the following, we present Lemmas 6.4.2 which explains why the cutoff result 6.3.2 allows us to verify the strong completeness property of the failure detector of [CT96] under synchrony by model checking instances of size 2.

**Proposition 6.4.21** Let $G_N = (C_N, T_N, R_N, G^N_0)$ be a global transition system of a symmetric point-to-point algorithm under the unrestricted model. Its indexes are $1..N$ for some $N \geq 1$. Let $i$ and $j$ be two indexes in the set $1..N$. Let $\mu_{\{i,j\}}$ be a first-order formula in which every predicate takes one of the forms: $Q_1(i)$, or $Q_2(j)$, or $Q_3(i,j)$, or $Q_4(j,i)$. The following conditions hold:

1. $FG \bigwedge_{i,j \in 1..N} \mu_{\{i,j\}}$ be an LTL formula.
2. $\bigwedge_{i,j \in 1..N} FG \mu_{\{i,j\}}$ be an LTL formula.
3. Let $\pi = \kappa_0 \kappa_1 \ldots$ be an admissible sequence of configurations in $G_N$. It follows $G_N, \pi \models FG \bigwedge_{i,j \in 1..N} \mu_{\{i,j\}}$ if and only if $G_N, \pi \models \bigwedge_{i,j \in 1..N} FG \mu_{\{i,j\}}$.

**Proof.** Points (1) and (2) hold by the definition of LTL formula (see [CJGK+18]). We prove Point (3) as follows.

$(\Rightarrow)$ Since $G_N, \pi \models FG \bigwedge_{i,j \in 1..N} \mu_{\{i,j\}}$, there exists $\ell_0 \geq 0$ such that for every $\ell \geq \ell_0$, we have $\kappa_\ell \models \bigwedge_{i,j \in 1..N} \mu_{\{i,j\}}$. Let $i_0$ and $j_0$ be two indexes in $1..N$. We have $\kappa_\ell \models \mu_{\{i_0,j_0\}}$ for every $\ell \geq \ell_0$. Hence, it follows $G_N, \pi \models FG \mu_{\{i_0,j_0\}}$. Because $i_0$ and $j_0$ are arbitrary indexes in $1..N$, it follows that $G_N, \pi \models \bigwedge_{i,j \in 1..N} FG \mu_{\{i,j\}}$.

$(\Leftarrow)$ Let $i_0$ and $j_0$ be two indexes in $1..N$. Since $G_N, \pi \models \bigwedge_{i,j \in 1..N} FG \mu_{\{i,j\}}$, it follows that $G_N, \pi \models FG \mu_{\{i_0,j_0\}}$. Therefore, there exists $\ell_0^\prime \geq 0$ such that $\kappa_\ell \models \mu_{\{i_0,j_0\}}$ for every $\ell \geq \ell_0^\prime$. Let $\ell = \text{max}\{\ell_i^\prime : i \in 1..N \land j \in 1..N\}$ where $\text{max}$ is a function to pick a maximum number in a finite set of natural numbers. It follows that $\kappa_\ell \models \mu_{\{i,j\}}$ for every $\ell \geq \ell_0$, for every $i, j \in 1..N$. Therefore, $G_N, \pi \models FG \bigwedge_{i,j \in 1..N} \mu_{\{i,j\}}$.

**Lemma 6.4.2** Let $G_2$ and $G_N$ be two global transition systems of a symmetric point-to-point algorithm such that:
6.4. Detailed Proofs for Cutoff Results in the Model of the Global Distributed Systems

1. These systems $G_2$ and $G_N$ have $2$ and $N$ processes respectively where $N \geq 2$.
2. All processes $G_2$ and $G_N$ follow Algorithm 1.1.
3. The model of computation of these systems is under the unrestricted model.

Let $\Pi$ be a set of indexes, and $\mu(\Pi)$ denote the strong completeness property in which the set of process indexes is $\Pi$, i.e.

$$\mu(\Pi) \triangleq \text{F G} (\forall p, q \in \Pi : (\text{Correct}(p) \land \neg \text{Correct}(q)) \Rightarrow \text{Suspected}(p, q))$$

Let $\text{Path}_2$ and $\text{Path}_N$ be sets of all admissible sequences of configurations in $G_2$ and in $G_N$, respectively. Then, it holds:

$$\forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu(1..N)$$

$$\iff \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \mu(1..2)$$

**Proof.** To keep the presentation simple, let $\nu(p, q)$ be a predicate such that $\nu(p, q) \triangleq (\text{Correct}(p) \land \neg \text{Correct}(q))$. Let $\pi_N$ be an admissible sequence of configurations in $G_N$. We have $G_N, \pi_N \models \mu(\Pi)$ if and only if $G_N, \pi_N \models \text{F G} (\forall p, q \in 1..N : \nu(p, q) \Rightarrow \text{Suspected}(p, q))$. It follows

$$G_N, \pi_N \models \mu(1..N)$$

$$\iff G_N, \pi_N \models \text{F G} (\bigwedge_{p,q \in 1..N} \nu(p, q) \Rightarrow \text{Suspected}(p, q))$$

$$\iff G_N, \pi_N \models \bigwedge_{p,q \in 1..N} \text{F G} (\nu(p, q) \Rightarrow \text{Suspected}(p, q)) \quad (\text{by Proposition } 6.4.21)$$

The last formula is equivalent to

$$G_N, \pi_N \models \bigwedge_{p,q \in 1..N} \text{F G} (\nu(p, q) \Rightarrow \text{Suspected}(p, q))$$

$$\land G_N, \pi_N \models \bigwedge_{p,q \in 1..N} \text{F G} (\nu(p, q) \Rightarrow \text{Suspected}(p, q))$$

For every $p, q \in 1..N$, if $p = q$, then $(\text{Correct}(p) \land \neg \text{Correct}(q)) = \bot$ (*). Hence, it follows that

$$G_N, \pi_N \models \bigwedge_{p \neq q, p,q \in 1..N} \text{F G} (\nu(p, q) \Rightarrow \text{Suspected}(p, q))$$

$$\iff G_N, \pi_N \models \bigwedge_{p \neq q, p,q \in 1..N} \text{F G} (\nu(p, q) \Rightarrow \text{Suspected}(p, q))$$

It follows that $\forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu$ if and only if

$$\forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \bigwedge_{p \neq q, p,q \in 1..N} \text{F G} (\nu(p, q) \Rightarrow \text{Suspected}(p, q))$$

149
By Theorem 6.3.2, it follows
\[ \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu(1..N) \]
\[ \iff \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \bigwedge_{p \neq q} F G(\nu(p, q) \Rightarrow \text{Suspected}(p, q)) \]

By point (*), we have
\[ \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu(1..N) \]
\[ \iff \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \bigwedge_{p, q \in 1..2} F G(\nu(p, q) \Rightarrow \text{Suspected}(p, q)) \]

By Proposition 6.4.21, it follows
\[ \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu(1..N) \]
\[ \iff \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \mu(1..2) \]

Hence, Lemma 6.4.2 holds.

**Lemma 6.4.3** Let \( G_1 \) and \( G_2 \) and \( G_N \) be three global transition systems of a symmetric point-to-point algorithm such that:

1. These systems \( G_1 \) and \( G_2 \) and \( G_N \) have 1, 2 and \( N \) processes respectively.
2. All processes in \( G_1 \) and \( G_2 \) and \( G_N \) follow Algorithm 1.1.
3. Three sets \( \text{Path}_1 \) and \( \text{Path}_2 \) and \( \text{Path}_N \) be sets of admissibles sequences of configurations in \( G_1 \) and \( G_2 \) and \( G_N \), respectively.
4. The model of computation of these systems is under the unrestricted model.

Let \( \Pi \) be a set of process indexes, and \( \mu(\Pi) \) denote the eventually strong accuracy property in which the set of process indexes is \( \Pi \), i.e.,

\[ \mu(\Pi) \triangleq F G(\forall p, q \in \Pi: (\text{Correct}(p) \land \text{Correct}(q)) \Rightarrow \neg \text{Suspected}(p, q)) \]

It follows \( \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu(1..N) \) if and only if both \( \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \mu(1..2) \) and \( \forall \pi_1 \in \text{Path}_1 : G_1, \pi_1 \models \mu(1..1) \).

**Proof.** To keep the presentation simple, we define
\[ \nu(p, q) = (\text{Correct}(p) \land \text{Correct}(q)) \Rightarrow \neg \text{Suspected}(p, q) \]
By similar arguments in Proposition [6.4.2], we have \( \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \mu(1..N) \) is equivalent to the following conjunction

\[
\left( \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \bigwedge_{p,q \in 1..N} F G \nu(p, q) \right)
\]

\[
\land \left( \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \bigwedge_{p \neq q \in 1..N} F G \nu(p, q) \right)
\]

By Theorems [6.3.1] and [6.3.2], the above conjunction is equivalent to

\[
\left( \forall \pi_1 \in \text{Path}_1 : G_1, \pi_1 \models \bigwedge_{p,q \in 1..N} F G \nu(1, 1) \right)
\]

\[
\land \left( \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \bigwedge_{p,q \in 1..N} F G \nu(1, 2) \right)
\]

By Proposition [6.4.2], the above conjunction is equivalent to

\[
\left( \forall \pi_1 \in \text{Path}_1 : G_1, \pi_1 \models F G \bigwedge_{p,q \in 1..N} \nu(1, 1) \right)
\]

\[
\land \left( \forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models F G \bigwedge_{p,q \in 1..N} \nu(1, 2) \right)
\]

Therefore, Lemma [6.4.3] holds.

### 6.5 Cutoff Results in the Case of Unknown Time Bounds

In this section, we extend the above cutoff results on a number of processes (see Theorems [6.3.1] and [6.3.2]) for partial synchrony in case of unknown bounds \( \Delta \) and \( \Phi \). The extended results are formalized in Theorems [6.5.1] and [6.5.2]. It is straightforward to adapt our approach to other models of partial synchrony in [DLS88, C99].

**Theorem 6.5.1** Let \( \mathcal{A} \) be a symmetric point-to-point algorithm under partial synchrony with unknown bounds \( \Delta \) and \( \Phi \). Let \( G_1 \) and \( G_N \) be instances of \( \mathcal{A} \) with 1 and \( N \) processes respectively for some \( N \geq 1 \). Let \( \text{Path}_1 \) and \( \text{Path}_N \) be sets of all admissible sequences of configurations in \( G_1 \) and in \( G_N \) under partial synchrony, respectively. Let \( \omega_\{i\} \) be a LTL/X formula in which every predicate takes one of the forms: \( P_1(i) \) or \( P_2(i,i) \) where \( i \) is an index in 1..\( N \). It follows that:

\[
\left( \forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \bigwedge_{i \in 1..N} \omega_\{i\} \right) \Leftrightarrow \left( \forall \pi_1 \in \text{Path}_1 : G_1, \pi_1 \models \omega_\{1\} \right)
\]
Theorem 6.5.2 Let $A$ be a symmetric point–to–point algorithm under partial synchrony with unknown bounds $\Delta$ and $\Phi$. Let $G_2$ and $G_N$ be instances of $A$ with 2 and $N$ processes respectively for some $N \geq 2$. Let $\text{Path}_2$ and $\text{Path}_N$ be sets of all admissible sequences of configurations in $G_2$ and in $G_N$ under partial synchrony, respectively. Let $\psi_{\{i,j\}}$ be an LTL formula in which every predicate takes one of the forms: $Q_1(i)$, or $Q_2(j)$, or $Q_3(i,j)$, or $Q_4(j,i)$ where $i$ and $j$ are different indexes in $1..N$. It follows that:

$$(\forall \pi_N \in \text{Path}_N : G_N, \pi_N \models \bigwedge_{i,j \in 1..N} \psi_{\{i,j\}}) \iff (\forall \pi_2 \in \text{Path}_2 : G_2, \pi_2 \models \psi_{\{1,2\}})$$

Since the proofs of these theorems are similar, we here focus on only Theorem 6.5.2. The proof of Theorem 6.5.2 follows the approach in [EN95, TKW20], and is based on the following observations. Remind that Steps 1 and 2 are already proved in Section 6.3.

1. The global transition system and the desired property are symmetric.

2. Let $G_2$ and $G_N$ be two instances of a symmetric point-to-point algorithm with 2 and $N$ processes, respectively. We have that two instances $G_2$ and $G_N$ are trace equivalent under a set of predicates in the desired property.

3. We will now discuss that the constraints maintain partial synchrony. Let $\pi_N$ be an execution in $G_N$. By applying the index projection to $\pi_N$, we obtain an execution $\pi_2$ in $G_2$. If partial synchrony constraints $\text{(PS1)}$ and $\text{(PS2)}$ – defined in Section 6.2.3 – hold on $\pi_N$, these constraints also hold on $\pi_2$. This result is proved in Lemma 6.5.1.

4. Let $\pi_2$ be an execution in $G_2$. We construct an execution $\pi_N$ in $G_N$ based on $\pi_2$ such that all processes $3..N$ crash from the beginning, and $\pi_2$ is an index projection of $\pi_N$ (defined in Section 6.3.2). For instance, see Figure 6.2. If partial synchrony constraints $\text{(PS1)}$ and $\text{(PS2)}$ – defined in Section 6.2.3 – hold on $\pi_2$, these constraints also hold on $\pi_N$. This result is proved in Lemma 6.5.2.

Lemma 6.5.1 Let $A$ be a symmetric point–to–point algorithm under partial synchrony with unknown bounds $\Delta$ and $\Phi$. Let $G_2$ and $G_N$ be instances of $A$ with 2 and $N$ processes, respectively, for some $N \geq 2$. Let $\text{Path}_2$ and $\text{Path}_N$ be sets of all admissible sequences of configurations in $G_2$ and in $G_N$ under partial synchrony, respectively. Let $\pi_N = \kappa_N^0 \kappa_N^1 \kappa_N^2 \ldots$ be an admissible sequence of configurations in $G_N$. Let $\pi_2 = \kappa_2^0 \kappa_2^1 \kappa_2^2 \ldots$ be a sequence of configurations in $G_2$ such that $\kappa_k^2$ be an index projection of $\kappa_k^N$ on indexes $\{1,2\}$ for every $k \geq 0$. It follows that

(a) Constraint $\text{(PS1)}$ on message delay holds on $\pi_2$.

(b) Constraint $\text{(PS2)}$ on the relative speed of processes holds on $\pi_2$. 

152
Proof. Recall that the index projection is defined in Section 6.3.2. In the following, we denote $p^2$ and $p^N$ two processes such that they have the same index, and $p^2$ is a process in $G_2$, and $p^N$ is a process in $G_N$. We prove Lemma 6.5.1 by contradiction.

(a) Assume that Constraint [PS1] does not hold on $\pi^2$. Hence, there exist a time $\ell > 0$, and two processes $s^2, r^2 \in 1..2$ in $G_2$ such that after $r^2$ executes Receive at a time $\ell$, there exists an old message in a message buffer from process $s^2$ to process $r^2$. By the definition of the index projection, for every $k \geq 0$, we have that:

- Let $p^N, q^N \in \{1, 2\}$ be two processes in $G_N$, and $p^2, q^2$ be corresponding processes in $G_2$. For every $k \geq 0$, two message buffers from process $p^2$ to process $q^2$ in $\kappa^2_k$, and from process $p^N$ to process $q^N$ in $\kappa^N_k$ are the same.

- Let $p^N \in \{1, 2\}$ be a process in $G_N$, and $p^2$ be a corresponding process in $G_2$. For every $k \geq 0$, process $p^2$ takes an action $act$ in configuration $\kappa^2_k$ if and only if process $p^N$ takes the same action in configuration $\kappa^N_k$.

It implies that process $r^N$ in $G_N$ also executes Receive at a time $\ell$, and there exists an old message in a buffer from process $s^N$ to process $r^N$. Contradiction.

(b) By applying similar arguments in case (a).

Lemma 6.5.2 Let $A$ be a symmetric point-to-point algorithm under partial synchrony with unknown bounds $\Delta$ and $\Phi$. Let $G_2$ and $G_N$ be instances of $A$ with 2 and $N$ processes, respectively, for some $N \geq 2$. Let $Path_2$ and $Path_N$ be sets of all admissible sequences of configurations in $G_2$ and in $G_N$ under partial synchrony, respectively. Let $\pi^2 = \kappa^2_0 \kappa^2_1 \ldots$ be an admissible sequences of configurations in $G_2$. Let $\pi^N = \kappa^N_0 \kappa^N_1 \ldots$ be a sequence of configurations in $G_N$ such that (i) every process $p \in 3..N$ crashes from the beginning, and (ii) $\kappa^2_k$ be an index projection of $\kappa^N_k$ on indexes $\{1, 2\}$ for every $k \geq 0$. It follows that

(a) Constraint [PS1] on message delay holds on $\pi^N$.

(b) Constraint [PS2] on the relative speed of processes holds on $\pi^N$.

Proof. By applying similar arguments in the proof of Lemma 6.5.1 and the facts that every process $p \in 3..N$ crashes from the beginning, and that $\kappa^2_k$ be an index projection of $\kappa^N_k$ on indexes $\{1, 2\}$ for every $k \geq 0$.

6.6 Experiments

To demonstrate the feasibility of our approach, we specified the failure detector [CT96] in TLA+ [Lam02]2. Our specification follows the model of computation in Section 6.2. It is close to the pseudo-code in Section 1.1 except that these tasks are organized in

2Our specification is available at [github.com/banhday/netys20.git](http://github.com/banhday/netys20.git)
6. Cutoffs for Symmetric Point-to-point Distributed Algorithms

We ran the following experiments on a laptop with a core i7-6600U CPU and 16GB DDR4. We checked benchmarks with TLC in the TLA+ Toolbox version 1.6.0 [Mic] and APALACHE version 0.5.2 described in Chapter 3. Table 6.1 presents the results in model checking the failure detectors [CT96] in the synchronous model. From the theoretical viewpoint, an instance with $N = 1$ is necessary, but we show only interesting cases with $N \geq 2$ in Table 6.1. (We did check an instance with $N = 1$, and there are no errors in this instance.) The strong accuracy property is the following safety property:

$$G(\forall p, q \in 1..N: (Correct(p) \land Correct(q)) \Rightarrow \neg Suspected(p, q)).$$

The column “depth” shows the maximum execution length used by our tool as well as the maximum depth reached by TLC while running breadth-first search. For the second and forth benchmarks, we used the diameter bound that was reported by TLC, which does exhaustive state exploration. Hence, the verification results with APALACHE are complete. The abbreviation “TO” means timeout of 10 hours. The inductive invariant is on the transition $\sim$, and contains type invariants, constraints on the age of in-transit messages, and constraints on when a process executes a task. This inductive invariant does not imply the safety properties of the Chandra and Toueg failure detector.
6.7 Summary

We have introduced the class of symmetric point-to-point algorithms that capture some well-known algorithms, e.g. failure detectors. The symmetric point-to-point algorithms enjoy the cutoff property. We have shown that checking properties of the form $\omega(i)$ has a cutoff of 1, and checking properties of the form $\psi(i, j)$ has a cutoff of 2 where $\omega(i)$ is an LTL\X formula whose predicates inspect only variables with a process index $i$, and $\psi(i, j)$ is an LTL\X formula whose predicates inspect only variables with two different process indexes $i \neq j$. We demonstrated the feasibility of our approach by specifying and model checking the Chandra and Toueg failure detector under synchrony with two model checkers TLC and APALACHE.

In the following chapter, these results are a cornerstone to verify the failure detector in case of unknown time bounds. By these results, we need to check only its instances with two processes and arbitrary bounds $\Delta$ and $\Phi$. 
A Case Study on Parametric Verification of Failure Detectors

In this chapter, we present a case study on formal verification of both safety and liveness of the Chandra and Toueg failure detector that is based on partial synchrony. To this end, we specify the algorithm and the partial synchrony assumptions in three frameworks: TLA+, IVy, and counter automata. Importantly, we tune our modeling to use the strength of each method: (1) We are using counters to encode message buffers with counter automata, (2) we are using first-order relations to encode message buffers in IVy, and (3) we are using both approaches in TLA+. By running the tools for TLA+ (TLC and APALACHE) and counter automata (FAST), we demonstrate safety for fixed time bounds. This helped us to find the inductive invariants for fixed parameters, which we used as a starting point for the proofs with IVy. By running IVy, we prove safety for arbitrary time bounds. Moreover, we show how to verify liveness of the failure detector by reducing the verification problem to safety verification. Thus, both properties are verified by developing inductive invariants with IVy. We conjecture that correctness of other partially synchronous algorithms may be proven by following the presented methodology.

This chapter presents an extended version of the paper at FORTE’21 [TKW21a] and partially of the journal paper at LMCS’23 [TKW23].

7.1 Overview

Verification techniques for distributed algorithms usually focus on two models of computation: synchrony [SKWZ19] and asynchrony [KLVW17a, KLVW17b]. Synchrony is hard to implement in real systems, while many basic problems in fault-tolerant distributed computing are unsolvable in asynchrony.
Partial synchrony lies between synchrony and asynchrony, and escapes their shortcomings. To guarantee liveliness properties, proof-of-stake blockchains [BKM18, YMR+19] and distributed algorithms [CT96, BCG20] assume time constraints under partial synchrony. That is, the existence of bounds $\Delta$ on message delay, and $\Phi$ on the relative speed of processes after some time point. This combination makes partially synchronous algorithms parametric in time bounds. While partial synchrony is important for system designers, it is challenging for verification.

We thus investigate verification of distributed algorithms under partial synchrony, and start with the specific class of failure detectors: a Chandra and Toueg failure detector [CT96] described in Section 1.2. In Chapter 6, we proved the cutoff results of the failure detector that allows us to verify this algorithm by checking only instances with two processes. Now we do parametric verification of both safety and liveness of the Chandra and Toueg failure detector in case of unknown bounds $\Delta$ and $\Phi$. In this case, both $\Delta$ and $\Phi$ are arbitrary, and the constraints on message delay and the relative speeds hold in every execution from the start.

To this end, we first introduce encoding techniques to efficiently specify the failure detector based on our cutoff results. These techniques can tune our modeling to use the strengths of the tools: FAST, IVy, and model checkers for TLA⁺. Second, we demonstrate how to reduce the liveness properties Eventually Strong Accuracy, and Strong Completeness to safety properties. Next, we check the safety property Strong Accuracy, and the mentioned liveness properties on instances with fixed parameters by using FAST, and model checkers for TLA⁺. Finally, to verify cases of arbitrary bounds $\Delta$ and $\Phi$, we find and prove inductive invariants of the failure detector with the interactive theorem prover IVy. We reduce the liveness properties to safety properties by applying the mentioned techniques. While our specifications are not in the decidable theories that IVy supports, IVy requires no additional user assistance to prove most of our inductive invariants.

**Structure.** Our encoding technique is presented in Section 7.2. In Section 7.3, we present how to reduce the mentioned liveness properties to safety ones. Experiments for small $\Delta$ and $\Phi$ are described in Section 7.4. IVy proofs for parametric $\Delta$ and $\Phi$ are discussed in Section 7.5.

### 7.2 Encoding the Chandra and Toueg Failure Detector

In this section, we first discuss why it is sufficient to verify the failure detector by checking a system with only one sender and one receiver by applying the cutoffs presented in Section 6.5. Next, we introduce two approaches to encoding the message buffer, and an abstraction of in-transit messages that are older than $\Delta$ time-units. Finally, we present how to encode the relative speed of processes with counters over natural numbers. These techniques allow us to tune our models to the strength of the verification tools: FAST, IVy, and model checkers for TLA⁺.
7.2. Encoding the Chandra and Toueg Failure Detector

7.2.1 The System with One Sender and One Receiver

The cutoff results in Section 6.5 allow us to verify the Chandra and Toueg failure detector under partial synchrony by checking only instances with two processes. In the following, we discuss the model with two processes, and formalize the properties with two-process indexes. By process symmetry, it is sufficient to verify Strong Accuracy, Eventually Strong Accuracy, and Strong Completeness by checking the following properties.

\[ G((\text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \neg \text{Suspected}(2, 1)) \]  
\[ FG((\text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \neg \text{Suspected}(2, 1)) \]  
\[ FG((\neg \text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \text{Suspected}(2, 1)) \]

We can take a further step towards facilitating verification of the failure detector. First, every process typically has a local variable to store messages that it needs to send to itself, instead of using a real communication channel. Hence, we can assume that there is no delay for these messages, and that each correct process never suspects itself. Second, local variables in Algorithm 1.1 are arrays whose elements correspond one-to-one with a remote process, e.g., \textit{timeout}[2, 1] and \textit{suspected}[2, 1]. Third, communication between processes is point-to-point. When this is not the case, one can use cryptography to establish one-to-one communication. Hence, reasoning about Properties 7.1–7.3 requires no information about messages from process 1 to itself, local variables of process 1, and messages from process 2.

Due to the above characteristics, it is sufficient to consider process 1 as a sender, and process 2 as a receiver. In detail, the sender follows Task 1 in Algorithm 1.1, but does nothing in Task 2 and Task 3. The sender does not need the initialization step, and local variables \textit{suspected} and \textit{timeout}. In contrast, the receiver has local variables corresponding to the sender, and follows only the initialization step, and Task 2, and Task 3 in Algorithm 1.1. The receiver can increase its waiting time in Task 1, but does not send any message.

7.2.2 Encoding the Message Buffer

Algorithm 1.1 assumes unbounded message buffers between processes that produce an infinite state space. Moreover, a sent message might be in-transit for a long time before it is delivered. We first introduce two approaches to encode the message buffer based on a logical predicate, and a counter over natural numbers. The first approach works for TLA$^+$ and IVy, but not for counter automata (FAST). The latter is supported by all mentioned tools, but it is less efficient as it requires more transitions. Then, we present an abstraction of in-transit messages that are older than $\Delta$ time-units. This technique reduces the state space, and allows us to tune our models to the strength of the verification tools.
7. A Case Study on Parametric Verification of Failure Detectors

Encoding the message buffer with a predicate.

In Algorithm 1.1, only “alive” messages are sent, and the message delivery depends only on the age of in-transit messages. Moreover, the computation of the receiver does not depend on the contents of its received messages. Hence, we can encode a message buffer by using a logical predicate \( \text{existsMsgOfAge}(x) \). For every \( k \geq 0 \), predicate \( \text{existsMsgOfAge}(k) \) refers to whether there exists an in-transit message that is \( k \) time-units old. The number 0 refers to the age of a fresh message in the buffer.

It is convenient to encode the message buffer’s behaviors in this approach. For instance, Formulas 7.4 and 7.5 show constraints on the message buffer when a new message is sent:

\[
\begin{align*}
\text{existsMsgOfAge'}(0) & \quad (7.4) \\
\forall x \in \mathbb{N}. x > 0 \Rightarrow \text{existsMsgOfAge'}(x) = \text{existsMsgOfAge}(x) & \quad (7.5)
\end{align*}
\]

where \( \text{existsMsgOfAge'} \) refers to the value of \( \text{existsMsgOfAge} \) in the next state. Formula (7.4) implies that a fresh message has been added to the message buffer. Formula (7.5) ensures that other in-transit messages are unchanged.

Another example is the relation between \( \text{existsMsgOfAge} \) and \( \text{existsMsgOfAge'} \) after the message delivery. This relation is formalized with Formulas 7.6–7.9. Formula 7.6 requires that there exists an in-transit message in \( \text{existsMsgOfAge} \) that can be delivered. Formula 7.7 ensures that no old messages are in transit after the delivery. Formula 7.8 guarantees that no message is created out of thin air. Formula 7.9 implies that at least one message is delivered.

\[
\begin{align*}
\exists x \in \mathbb{N}. \text{existsMsgOfAge}(x) & \quad (7.6) \\
\forall x \in \mathbb{N}. x \geq \Delta \Rightarrow \neg\text{existsMsgOfAge'}(x) & \quad (7.7) \\
\forall x \in \mathbb{N}. \text{existsMsgOfAge'}(x) \Rightarrow \text{existsMsgOfAge}(x) & \quad (7.8) \\
\exists x \in \mathbb{N}. \text{existsMsgOfAge'}(x) \neq \text{existsMsgOfAge}(x) & \quad (7.9)
\end{align*}
\]

This encoding works for TLA\(^+\) and IVy, but not for FAST, because the input language of FAST does not support functions.

Encoding the message buffer with a counter.

In the following, we present an encoding technique for the buffer that can be applied in all tools TLA\(^+\), IVy, and FAST. This approach encodes the message buffer with a counter \( \text{buf} \) over natural numbers. The \( k^{th} \) bit refers to whether there exists an in-transit message with \( k \) time-units old.

In this approach, message behaviors are formalized with operations in Presburger arithmetic. For example, assume \( \Delta > 0 \), we write \( \text{buf}' = \text{buf} + 1 \) to add a fresh message in the buffer. Notice that the increase of \( \text{buf} \) by 1 turns on the \( 0^{th} \) bit, and keeps the other bits unchanged.
7.2. Encoding the Chandra and Toueg Failure Detector

To encode the increase of the age of every in-transit message by 1, we simply write \( \text{buf}' = \text{buf} \times 2 \). Assume that we use the least significant bit (LSB) first encoding, and the left-most bit is the \( 0^{th} \) bit. By multiplying \( \text{buf} \) by 2, we have updated \( \text{buf}' \) by shifting to the right every bit in \( \text{buf} \) by 1. For example, Figure 7.1 demonstrates the message buffer after the increase of message ages in case of \( \text{buf} = 6 \). We have \( \text{buf}' = \text{buf} \times 2 = 12 \). It is easy to see that the 1\( st \) and 2\( nd \) bits in \( \text{buf} \) are on, and the 2\( nd \) and 3\( rd \) bits in \( \text{buf}' \) are on.

Recall that Presburger arithmetic does not allow one to divide by a variable. Therefore, to guarantee the constraint in Formula 7.8 we need to enumerate all constraints on possible values of \( \text{buf} \) and \( \text{buf}' \) after the message delivery. For example, assume \( \text{buf} = 3 \), and \( \Delta = 1 \). After the message delivery, \( \text{buf}' \) is either 0 or 1. If \( \text{buf} = 2 \) and \( \Delta = 1 \), \( \text{buf}' \) must be 0 after the message delivery. Importantly, the number of transitions for the message delivery depends on the value of \( \Delta \).

To avoid the enumeration of all possible cases, Formula 7.8 can be rewritten with bit-vector arithmetic. However, bit-vector arithmetic are currently not supported in all verification tools TLA\(^+\), FAST, and IVy.

The advantage of this encoding is that when bound \( \Delta \) is fixed, every constraint in the system behaviors can be rewritten in Presburger arithmetic. Thus, we can use FAST, which accepts constraints in Presburger arithmetic. To specify cases with arbitrary \( \Delta \), the user can use TLA\(^+\) or IVy.

**Abstraction of old messages.**

Algorithm 1.1 assumes underlying unbounded message buffers between processes. Moreover, a sent message might be in transit for a long time before it is delivered. To reduce the state space, we develop an abstraction of in-transit messages that are older than \( \Delta \) time-units; we call such messages “old”. This abstraction makes the message buffer between the sender and the receiver bounded. In detail, the message buffer has a size of \( \Delta \). Importantly, we can apply this abstraction to two above encoding techniques for the message buffer.

In partial synchrony, if process \( p \) executes Receive at some time point from the Global Stabilization Time, every old message sent to \( p \) will be delivered immediately. Moreover, the computation of a process in Algorithm 1.1 does not depend on the content of received messages. Hence, instead of tracking all old messages, our abstraction keeps only one old message that is \( \Delta \) time-units old, does not increase its age, and throws away other old messages.

<table>
<thead>
<tr>
<th>Age indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>⋮</th>
</tr>
</thead>
<tbody>
<tr>
<td>Messages in ( \text{buf} )</td>
<td>16</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Figure 7.1: The message buffer after increasing message ages in case of \( \text{buf} = 6 \)
7. A Case Study on Parametric Verification of Failure Detectors

<table>
<thead>
<tr>
<th>Age indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) Messages in buf</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age indexes</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) Messages in buf</td>
<td>[ ]</td>
<td>[ ]</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

Figure 7.2: The increase of message ages with the abstraction of old messages. In the case (a), we have $\Delta = 2$, $buf = 6$, and $buf' = 4$. In the case (b), we have $\Delta = 2$, $buf = 5$, and $buf' = 6$.

```
1: if $buf < 2\Delta$ then $buf' \leftarrow buf \times 2$
2: else
3: if $buf \geq 2\Delta + 2\Delta - 1$ then $buf' \leftarrow buf \times 2 - 2\Delta + 1$
4: else $buf' \leftarrow buf \times 2 - 2\Delta + 1 + 2\Delta$
```

Figure 7.3: Encoding the increase of message ages with a counter $buf$, and the abstraction of old messages.

In the following, we discuss how to integrate this abstraction into the encoding techniques of the message buffer. We demonstrate our ideas by showing the pseudo code of the increase of message ages. It is straightforward to adopt this abstraction to the message delivery, and to the sending of a new message.

Figure 7.2(a) presents the increase of message ages with this abstraction in a case of $\Delta = 2$, and $buf = 6$. Unlike Figure 7.1, there exists no in-transit message that is 3 time-units old in Figure 7.2(a). Moreover, the message buffer in Figure 7.2(a) has a size of 3. In addition, $buf'$ has only one in-transit message that is 2 time-units old. We have $buf' = 4$ in this case. Figure 7.2(b) demonstrates another case of $\Delta = 2$, $buf = 5$, and $buf' = 6$.

Formally, Figure 7.3 presents the pseudo code of the increase of message ages that is encoded with a counter $buf$, and the abstraction of old messages. There are three cases. In the first case (Line 1), there exist no old messages in $buf$, and we simply set $buf' = buf \times 2$. In other cases (Lines 3 and 4), $buf$ contains an old message. Figure 7.2(a) demonstrates the second case (Line 3). We subtract $2\Delta + 1$ to remove an old message with $\Delta + 1$ time-units old from the buffer. Figure 7.2(b) demonstrates the third case (Line 4). In the third case, we also need to remove an old message with $\Delta + 1$ time-units old from the buffer. Moreover, we need to put an old message with $\Delta$ time-units old to the buffer by adding $2\Delta$.

Now we discuss how to integrate the abstraction of old messages in the encoding of the message buffer with a predicate. Formulas 7.10–7.13 present the relation between $\text{existsMsgOfAge}$ and $\text{existsMsgOfAge}'$ when message ages are increased by 1, and this abstraction is applied. Formula 7.10 ensures that no fresh message will be added to
7.3. Reduce Liveness Properties to Safety Properties

\( \exists \text{MsgOfAge}' \). Formula \( \text{existsMsgOfAge}' \) ensures that the age of every message that is until \((\Delta - 2)\) time-units old will be increased by 1. Formulas \( \text{Formula 7.12} \) and \( \text{Formula 7.13} \) are introduced by this abstraction. Formula \( \text{Formula 7.12} \) implies that if there exists an old message or a message with \((\Delta - 1)\) time-units old in \( \exists \text{MsgOfAge} \), there will be an old message that is \( \Delta \) time-units old in \( \exists \text{MsgOfAge}' \). Formula \( \text{Formula 7.13} \) ensures that there exists no message that is older than \( \Delta \) time-units old.

\[
\begin{align*}
\neg \exists \text{MsgOfAge}'(0) & \quad (7.10) \\
\forall x \in \mathbb{N}. (0 \leq x \leq \Delta - 2) \Rightarrow \exists \text{MsgOfAge}'(x + 1) = \exists \text{MsgOfAge}(x) & \quad (7.11) \\
\exists \text{MsgOfAge}'(\Delta) = \exists \text{MsgOfAge}(\Delta) \lor \exists \text{MsgOfAge}(\Delta - 1) & \quad (7.12) \\
\forall x \in \mathbb{N}. x > \Delta \Rightarrow \exists \text{MsgOfAge}'(x) = \bot & \quad (7.13)
\end{align*}
\]

7.2.3 Encoding the Relative Speed of Processes

Recall that we focus on the case of unknown bounds \( \Delta \) and \( \Phi \). In this case, every correct process must take at least one step in every contiguous time interval containing \( \Phi \) time-units [DLSSS].

To maintain this constraint on executions generated by the verification tools, we introduced two additional control variables \( s\text{Timer} \) and \( r\text{Timer} \) for the sender and the receiver, respectively. These variables work as timers to keep track of how long a process has not taken a step, and when a process can take a step. Since these timers play similar roles, we here focus on \( r\text{Timer} \). In our encoding, only the global system can update \( r\text{Timer} \). To schedule the receiver, the global systems non-deterministically executes one of two actions in the sub-round Schedule: (i) resets \( r\text{Timer} \) to 0, and (ii) if \( r\text{Timer} < \Phi \), increases \( r\text{Timer} \) by 1. In other sub-rounds, the value of \( r\text{Timer} \) is unchanged. Moreover, the receiver must take a step whenever \( r\text{Timer} = 0 \).

7.3 Reduce Liveness Properties to Safety Properties

To verify the liveness properties Eventually Strong Accuracy and Strong Completeness with IVy, we first need to reduce them to safety properties. Intuitively, these liveness properties are bounded; therefore, they become safety ones. In the following, we explain how to do that.

7.3.1 Eventually Strong Accuracy

By cutoffs discussed in Section 6.5, it is sufficient to verify Eventually Strong Accuracy on the Chandra and Toueg failure detector by checking the following property on instances with 2 processes.

\[
\mathbf{FG}((\text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \neg \text{Suspected}(2, 1)) \quad (7.14)
\]

163
where process 1 is the sender and process 2 is the receiver.

In the following, we present how to reduce Formula 7.14 to a safety property. Our reduction is based on the following observations:

1. Fairness (Line 5 in Algorithm 7.1): correct processes send “alive” infinitely often.

2. The reliable communication (Constraint (TC1)): Let \( \text{rcv}_\text{msg}_\text{from}(2,1) \) be a predicate that refers to whether process 2 receives a message from process 1. If processes \( p \) and \( q \) are always correct, then it holds \( \text{GF} \text{rcv}_\text{msg}_\text{from}(2,1) \).

3. Transition invariant: Let \( \psi_1(2,1) \) be a predicate such that:
   \[
   \psi_1(2,1) \triangleq \text{rcv}_\text{msg}_\text{from}(2,1) \land \text{Correct}(1) \land \text{Correct}(2) \land \text{Suspected}(2,1)
   \]

   Then, the following property is a transition invariant.
   \[
   \text{GF}(\psi_1(p,q) \Rightarrow \text{timeout}[2,1] = \text{timeout}[2,1] + 1) \tag{7.15}
   \]

Points 1–3 implies that if \( \text{timeout}[2,1] \) is always less than some constant in an arbitrary execution path, then Formula 7.14 holds in this path.

Now we discuss why \( \text{timeout}[2,1] \) does not keep increasing forever if processes 1 and 2 are correct. To that end, we find a specific guard \( g \) for \( \text{timeout}[2,1] \) such that if \( \text{timeout}[2,1] \geq g \) and the sender is correct, then the receiver waits for the sender in less than \( g \) time-units. Moreover, the value of \( g \) depends only on the values of \( \Delta \) and \( \Phi \). Hence, it is sufficient to verify Formula 7.14 by checking Formula 7.16.

\[
\text{GF}(\text{timeout}[2,1] \geq g \Rightarrow ((\text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \neg\text{Suspected}(2,1))) \tag{7.16}
\]

### 7.3.2 Strong Completeness

By cutoffs discussed in Section 6.5, it is sufficient to verify Strong Completeness on the Chandra and Toueg failure detector by checking the following property on instances with 2 processes.

\[
\text{FG}((\neg\text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \text{Suspected}(2,1)) \tag{7.17}
\]

Notice that in partial synchrony, every sent message is eventually delivered. Hence, after the sender crashes, the receiver eventually receives nothing from the sender. To reduce Formula 7.17 to a safety property, we first introduced a ghost variable \( h_{\text{LFSC}} \) to measure for how long the sender has crashed. \( h_{\text{LFSC}} \) is set to 0 when the sender crashes. After that, \( h_{\text{LFSC}} \) is increased by 1 in every global step if the receiver has not suspected the

---

1 As the default-value of \( \text{timeout}[2,1] \) is a parameter, \( \text{timeout}[2,1] \) might be greater than \( g \) from the initialization.
7.4. Experiments for Small Δ and Φ

crashed sender. Let \( \psi_2(2, 1) \) denote the constraint: \( \psi_2(2, 1) \triangleq \neg \text{Correct}(1) \land \text{Correct}(2) \land \neg \text{Suspected}(2, 1) \). Then, the following property is a transition invariant.

\[
G(\psi_2(p, q) \Rightarrow h\text{LFSC}' = h\text{LFSC} + 1) \quad (7.18)
\]

By Formula 7.18 if \( h\text{LFSC} \) is always less than some constant in an arbitrary execution path, then Formula 7.17 holds in this path.

Now we show that \( h\text{LFSC} \) cannot keep increasing forever. To that end, we find a specific guard \( g' > 0 \) for \( h\text{LFSC} \) such that if \( h\text{LFSC} = g' \), then the receiver suspects the sender. It implies that \( h\text{LFSC} \) is unchanged. Moreover, the value of \( g' \) depends only on the values of \( \Delta \) and \( \Phi \). Hence, it is sufficient to verify Formula 7.17 by checking Formula 7.19.

\[
G (h\text{LFSC} = g' \Rightarrow (\neg \text{Correct}(1) \land \text{Correct}(2)) \Rightarrow \text{Suspected}(2, 1)) \quad (7.19)
\]

7.4 Experiments for Small Δ and Φ

In this section, we describe our experiments with TLA+ and FAST. We ran the following experiments on a virtual machine with Core i7-6600U CPU and 8GB DDR4. Our specifications can be found at [TKW].

7.4.1 Model Checkers for TLA+: TLC and APALACHE

We first use TLA+ [Lam02] to specify the failure detector with both encoding techniques for the message buffer, and the abstraction in Section 7.2. Then, we use the model checker TLC in the TLA+ Toolbox version 1.7.1 [YML99, Mic] and the model checker APALACHE version 0.15.0 [KKT19, Sys19] to verify instances with fixed bounds \( \Delta \) and \( \Phi \), and the GST \( T_0 = 1 \). This approach helps us to search constraints in inductive invariants in case of fixed parameters. The main reason is that counterexamples and inductive invariants in case of fixed parameters, e.g., \( \Delta \leq 1 \) and \( \Phi \leq 1 \), are simpler than in case of arbitrary parameters. Hence, if a counterexample is found, we can quickly analyze it, and change constraints in an inductive invariant candidate. We apply the counterexample-guided approach to find inductive strengthenings. After obtaining inductive invariants in small cases, we can generalize them for cases of arbitrary bounds, and check with theorem provers, e.g., IVy (Section 7.5).

TLA+ offers a rich syntax for sets, functions, tuples, records, sequences, and control structures [Lam02]. Hence, it is straightforward to apply the encoding techniques and the abstraction presented in Section 7.2 in TLA+. For example, Figure 7.4 represents a TLA+ action \( SSnd \) for sending a new message in case of \( \Delta > 0 \). Variables \( e\text{PC} \) and \( s\text{PC} \) are program counters for the environment and the sender, respectively. Line 1 is a precondition, and refers to that the environment is in subround Send. Lines 2–3 say that if the sender is active in subround Send, the counter \( buf' \) is increased by 1. Otherwise,
7. A Case Study on Parametric Verification of Failure Detectors

\[ SSnd = \Delta \land ePC = "SSnd" \]

1: 

\[ \text{if} (sTimer = 0 \land sPC = "SSnd") \]

2: 

\[ \text{then } buf' = buf + 1 \]

3: 

\[ \text{else unchanged } buf \]

4: 

\[ ePC' = "RNoSnd" \]

5: 

\[ \text{unchanged } sTimer, rTimer... \]

6: 

Figure 7.4: Sending a new message in TLA\(^+\) in case of \(\Delta > 0\)

\[ \text{Next} = \lor SSched \lor RSched \lor IncMsgAge \]

1: 

\[ \lor SSnd \lor RNoSnd \]

2: 

\[ \lor RRcv \]

3: 

\[ \lor RComp \]

Figure 7.5: The Next predicate for the next-state relation in TLA\(^+\)

two counters \(buf\) and \(buf'\) are the same (Line 4). Line 5 implies that the environment is still in the subround Send, but it is now the receiver's turn. Line 6 guarantees that other variables are unchanged in this action.

Figure 7.5 represents the next–state relation in TLA\(^+\). Line 1 describes actions in sub-round Schedule. The environment schedules the Sender, schedules the Receiver, and then increases message ages. Lines 2, 3, and 4 describes actions in sub-rounds Send, Receive, and Computation, respectively. The program counter \(ePC\) of the environment is used to ensure that every action is repeated periodically and in order.

Figure 7.6 represents how the environment schedules the Receiver in TLA\(^+\). Line 1 says that the current step is to schedule the Receiver, and Line 2 refers to the next action that is to increase message ages. Line 3 non-deterministically sets the Receiver active in the current global step. Lines 4–6 are to update the program counter \(rPC\) of the Receiver. The environment schedules the Sender, schedules the Receiver, and then increases message ages. Lines 7–8 non-deterministically sets the Receiver inactive in the current global step if the Receiver is not frozen in the last \(\Phi - 1\) global steps. Line 9 is to keep other variables unchanged.

Now we present the experiments with TLC and APALACHE. We used these tools to verify (i) the safety property Strong Accuracy, and (ii) an inductive invariant for Strong Accuracy, and (iii) an inductive invariant for a safety property reduced from the liveness property Strong Completeness in case of fixed bounds, and GST = 1 (initial stabilization). The structure of the inductive invariants verified here are very close to one in case of arbitrary bounds \(\Delta\) and \(\Phi\). While all parameters are assigned specific values in the inductive invariants of small instances, they have arbitrary values in the case of arbitrary...
7.4. Experiments for Small $\Delta$ and $\Phi$

1: $RSched \triangleq \wedge ePC = "RSched"
2: $\wedge ePC' = "IncMsgAge"
3: $\wedge \lor \wedge rTimer' = 0$
4: $\lor (rPC = "RNoSnd" \land rPC' = "RRcv")$
5: $\lor (rPC = "RRcv" \land rPC' = "RComp")$
6: $\lor (rPC = "RComp" \land rPC' = "RNoSnd")$
7: $\lor \wedge rTimer < \Phi - 1$
8: $\wedge rTimer' = rTimer + 1$
9: $\wedge \textit{UNCHANGED} \langle \textit{buf}, sTimer...\rangle$

Figure 7.6: The RSched predicate for scheduling the Receiver in TLA$^+$

Table 7.1 shows the results in verification of Strong Accuracy in case of the initial stabilization, and fixed bounds $\Delta$ and $\Phi$. Table 7.1 shows the experiments with the three tools TLC, APALACHE, and FAST. The column “#states” shows the number of distinct states explored by TLC. The column “#depth” shows the maximum execution length reached by TLC and APALACHE. The column “buf” shows how to encode the message buffer. The column “LOC” shows the number of lines in the specification of the system behaviors (without comments). The symbol “-” (minus) refers to that the experiments are intentionally missing since FAST does not support the encoding of the message buffer with a predicate. The abbreviation “pred” refers to the encoding of the message buffer with a predicate. The abbreviation “cntr” refers to the encoding of the message buffer with a counter. The abbreviation “TO” means a timeout of 6 hours. In these experiments, we initially set $\text{timeout} = 6 \times \Phi + \Delta$, and Strong Accuracy is satisfied. The experiments show that TLC finishes its tasks faster than the others, and APALACHE prefers the encoding of the message buffer with a predicate.

Table 7.2 summarizes the results in verification of Strong Accuracy with the tools TLC, APALACHE, and FAST in case of the initial stabilization, and small bounds $\Delta$ and $\Phi$, and initially $\text{timeout} = \Delta + 1$. Since $\text{timeout}$ is initialized with a too small value, there exists a case in which sent messages are delivered after the timeout expires. The tools reported an error execution where Strong Accuracy is violated. In these experiments, APALACHE is the winner. The abbreviation “TO” means a timeout of 6 hours. The meaning of other columns and abbreviations is the same as in Table 7.1.

Table 7.3 shows the results in verification of inductive invariants for Strong Accuracy and Strong Completeness with TLC and APALACHE in case of the initial stabilization, and slightly larger but fixed bounds $\Delta$ and $\Phi$, e.g., $\Delta = 20$ and $\Phi = 20$. The message buffer was encoded with a predicate in these experiments. In these experiments, inductive invariants
7. A Case Study on Parametric Verification of Failure Detectors

Table 7.1: Showing Strong Accuracy for fixed parameters.

<table>
<thead>
<tr>
<th>#</th>
<th>△</th>
<th>Φ</th>
<th>buf</th>
<th>TLC</th>
<th>APALACHE</th>
<th>FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time</td>
<td>#states</td>
<td>depth</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>pred</td>
<td>3s</td>
<td>10.2K</td>
<td>176</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cntr</td>
<td>3s</td>
<td>10.2K</td>
<td>176</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>pred</td>
<td>3s</td>
<td>16.6K</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cntr</td>
<td>3s</td>
<td>16.6K</td>
<td>183</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>5</td>
<td>pred</td>
<td>3s</td>
<td>44.7K</td>
<td>267</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td>cntr</td>
<td>3s</td>
<td>44.7K</td>
<td>267</td>
</tr>
</tbody>
</table>

Table 7.2: Violating Strong Accuracy for fixed parameters.

<table>
<thead>
<tr>
<th>#</th>
<th>△</th>
<th>Φ</th>
<th>buf</th>
<th>TLC</th>
<th>APALACHE</th>
<th>FAST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time</td>
<td>#states</td>
<td>depth</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>pred</td>
<td>1s</td>
<td>840</td>
<td>43</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cntr</td>
<td>1s</td>
<td>945</td>
<td>43</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>4</td>
<td>pred</td>
<td>2s</td>
<td>1.3K</td>
<td>48</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>cntr</td>
<td>2s</td>
<td>2.4K</td>
<td>56</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>20</td>
<td>pred</td>
<td>TO</td>
<td>22.1K</td>
<td>77</td>
</tr>
</tbody>
</table>

Table 7.3: Proving inductive invariants with TLC and APALACHE.

<table>
<thead>
<tr>
<th>#</th>
<th>△</th>
<th>Φ</th>
<th>Property</th>
<th>TLC</th>
<th>APALACHE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>time</td>
<td>#states</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>40</td>
<td>Strong Accuracy</td>
<td>33m</td>
<td>347.3M</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>10</td>
<td>Strong Completeness</td>
<td>44m</td>
<td>13.4M</td>
</tr>
</tbody>
</table>

hold, and APALACHE is faster than TLC in verifying them. In our experiment, we applied the counterexample-guided approach to manually find inductive strengthenings.

As one sees from the tables, APALACHE is fast at proving inductive invariants, and at finding a counterexample when a desired safety property is violated. TLC is a better option in cases where a safety property is satisfied.

In order to prove correctness of the failure detector in cases where parameters △ and Φ are arbitrary, the user can use the interactive theorem prover TLA⁺ Proof System (TLAPS) [CDLM10]. A shortcoming of TLAPS is that it does not provide a counterexample when an inductive invariant candidate is violated. Moreover, proving the failure detector with TLAPS requires more human effort than with IVy. Therefore, we provide IVy proofs in Section 7.5.
7.4.2 FAST

A shortcoming of the model checkers TLC and APALACHE is that parameters \( \Delta \) and \( \Phi \) must be fixed before running these tools. FAST is a tool designed to reason about safety properties of counter systems, i.e., automata extended with unbounded integer variables \([BLP06]\). If \( \Delta \) is fixed, and the message buffer is encoded with a counter, the failure detector becomes a counter system. We specified the failure detector in FAST, and made experiments with different parameter values to understand the limit of FAST: (i) the initial stabilization, and small bounds \( \Delta \) and \( \Phi \), and (ii) the initial stabilization, fixed \( \Delta \), but unknown \( \Phi \).

Figure 7.7 represents a FAST transition for sending a new message in case of \( \Delta > 0 \). Line 2 describes the (symbolic) source state of the transition, and region \( \text{incMsgAge} \) is a set of configurations in the failure detector that is reachable from a transition for increasing message ages. Line 3 mentions the (symbolic) destination state of the transition, and region \( \text{sSnd} \) is a set of configurations in the failure detector that is reachable from a transition named “SSnd_Active” for sending a new message. Line 4 represents the guard of this transition. Line 5 is an action. Every unprimed variable that is not written in Line 5 is unchanged.

The input language of FAST is based on Presburger arithmetics for both system and properties specification. Hence, we cannot apply the encoding of the message buffer with a predicate in FAST.

Tables 7.1 and 7.2 described in the previous subsection summarize the experiments with FAST, and other tools where all parameters are fixed. Moreover, we ran FAST to verify Strong Accuracy in case of the initial stabilization, \( \Delta \leq 4 \), and arbitrary \( \Phi \). FAST is a semi-decision procedure; therefore, it does not terminate on some inputs. Unfortunately, FAST could not prove Strong Accuracy in case of arbitrary \( \Phi \), and crashed after 30 minutes.

7.5 IVy Proofs for Parametric \( \Delta \) and \( \Phi \)

While TLC, APALACHE, and FAST can automatically verify some instances of the failure detector with fixed parameters, these tools cannot handle cases with unknown bounds \( \Delta \) and \( \Phi \). To overcome this problem, we specify and prove correctness of the
failure detector with the interactive theorem prover IVy \cite{Manshaei:2020}. In the following, we first discuss the encoding of the failure detector, and then present the experiments with IVy.

The encoding of the message buffer with a counter requires that bound \( \Delta \) is fixed. We here focus on cases where bound \( \Delta \) is unknown. Hence, we encode the message buffer with a predicate in our IVy specifications,

In IVy, we declare relation \( \text{existsMsgOfAge}(X : \text{num}) \). Type \( \text{num} \) is interpreted as integers. Since IVy does not support primed variables, we need an additional relation \( \text{tmpExistsMsgOfAge}(X : \text{num}) \). Intuitively, we first compute and store the value of \( \text{existsMsgOfAge} \) in the next state in \( \text{tmpExistsMsgOfAge} \), then copy the value of \( \text{tmpExistsMsgOfAge} \) back to \( \text{existsMsgOfAge} \). We do not consider the requirement of \( \text{tmpExistsMsgOfAge} \) as a shortcoming of IVy since it is still straightforward to transform the ideas in Section 7.2 to IVy.

Figure 7.1 represents how to add a fresh message in the message buffer in IVy. Line 1 means that \( \text{tmpExistsMsgOfAge} \) is assigned an arbitrary value. Line 2 guarantees the appearance of a fresh message. Line 3 ensures that every in-transit message in \( \text{existsMsgOfAge} \) is preserved in \( \text{tmpExistsMsgOfAge} \). Line 4 copies the value of \( \text{tmpExistsMsgOfAge} \) back to \( \text{existsMsgOfAge} \).

Algorithm 7.1 Adding a fresh message in IVy

1: \( \text{tmpExistsMsgOfAge}(X) := *; \)
2: \( \text{assume} \ \text{tmpExistsMsgOfAge}(0); \)
3: \( \text{assume} \ \forall X : \text{num}. \ 0 < X \rightarrow \text{existsMsgOfAge}(X) = \text{tmpExistsMsgOfAge}(X); \)
4: \( \text{existsMsgOfAge}(X) := \text{tmpExistsMsgOfAge}(X); \)

Importantly, our specifications are not in decidable theories supported by IVy. In Formula 7.11 the interpreted function \( \text{" + "} \) (addition) is applied to a universally quantified variable \( x \).

The standard way to check whether a safety property \( \text{Prop} \) holds in an IVy specification is to find an inductive invariant \( \text{IndInv} \) with \( \text{Prop} \), and to (interactively) prove that \( \text{IndInv} \) holds in the specification. To verify the liveness properties Eventually Strong Accuracy, and Strong Completeness, we reduced them into safety properties by applying a reduction technique in Section 7.3 and found inductive invariants containing the resulting safety properties reduced from the liveness properties. These inductive invariants are the generalization of the inductive invariants in case of fixed parameters that were found in the previous experiments.

Table 7.4 shows the experiments on verification of the failure detector with IVy in case of unknown \( \Delta \) and \( \Phi \). The symbol \( * \) refers to that the initial value of \( \text{timeout} \) is arbitrary. The column “\#lines” shows the number of lines of an inductive invariant, and the column “\#strengthening steps” shows the number of lines of strengthening steps that we provided for IVy. The meaning of other columns is the same as in Table 7.1 While our specifications are not in the decidable theories supported in IVy, our experiments
Table 7.4: Proving inductive invariants with IVy for arbitrary $\Delta$ and $\Phi$.

<table>
<thead>
<tr>
<th>#</th>
<th>Property</th>
<th>$\text{timeout}_{\text{init}}$</th>
<th>time</th>
<th>LOC</th>
<th>#line</th>
<th>$#$ strengthening steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Strong Accuracy</td>
<td>$= 6 \times \Phi + \Delta$</td>
<td>4s</td>
<td>183</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Eventually Strong Accuracy</td>
<td>$= \ast$</td>
<td>4s</td>
<td>186</td>
<td>35</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Strong Completeness</td>
<td>$= 6 \times \Phi + \Delta$</td>
<td>8s</td>
<td>203</td>
<td>111</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>$\geq 6 \times \Phi + \Delta$</td>
<td>22s</td>
<td>207</td>
<td>124</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$= \ast$</td>
<td>44s</td>
<td>207</td>
<td>129</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

show that IVy needs no user-given strengthening steps to prove most of our inductive invariants. Hence, it took us about 4 weeks to learn IVy from scratch, and to prove these inductive invariants.

The most important thing to prove a property satisfied in an IVy specification is to find an inductive invariant. Our inductive invariants use non-linear integers, quantifiers, and uninterpreted functions. (The inductive invariants in Table 7.4 are given in the repository [TKW].)

While IVy supports a liveness-to-safety reduction [PLSS17], this technique is not fully automated, and IVy still needs user-guided inductive invariant for reduced safety properties that may be different from those in Table 7.4. Moreover, IVy has not supported reasoning techniques for clocks. Therefore, we did not try the liveness-to-safety reduction of IVy.

It is straightforward to generalize the inductive invariants in Table 7.4 for partially synchronous models with known time bounds in [DLS88, CT96]. To reason about models with GST > 0, we need to find additional inductive strengthenings because the global system is under asynchrony before GST. Other partially synchronous models in [ABND+87] consider additional parameters, e.g., message order or point-to-point transmission that are out of scope of this paper.

7.6 Summary

We have presented verification of both safety and liveness of the Chandra and Toeg failure detector by using the verification tools: model checkers for TLA+ (APALACHE and TLC), counter automata (FAST), and the theorem prover IVy. To do that, we proved the cutoff results that can apply to the failure detector under partial synchrony in Chapter 6. In this chapter, we develop the encoding techniques to efficiently specify the failure detector, and to tune our models to use the strength of the mentioned tools. We verified safety in case of fixed parameters by running the tools TLC, APALACHE, and FAST. To cope with cases of arbitrary bounds $\Delta$ and $\Phi$, we reduced liveness properties to safety properties, and proved inductive invariants with desired properties in IVy. While our specifications are not in the decidable theories supported in IVy, our
7. A Case Study on Parametric Verification of Failure Detectors

experiments show that IVy needs no additional user assistance to prove most of our inductive invariants.

Modeling the failure detector in TLA$^+$ helped us understand and find inductive invariants in case of fixed parameters. Their structure is simpler but similar to the structure of parameterized inductive invariants. We found that the TLA$^+$ Toolbox [KLR19] has convenient features, e.g., Profiler and Trace Exploration. A strong point of IVy is in producing a counterexample quickly when a property is violated, even if all parameters are arbitrary. In contrast, FAST reports no counterexample in any case. Hence, debugging in FAST is very challenging.
Conclusions and Future Work

In this thesis, we developed symbolic verification techniques for TLA+ specifications. TLA+ is an expressive language based on untyped first-order logic and ZFC set theory. In TLA+, a specification is written as a logical formula. As a logic, TLA+ does not support assignments and other imperative statements that are used by model checkers to compute the successor states of a system state. To make symbolic reasoning for TLA+ efficiently, we developed an automatic technique to extract symbolic transitions in a specification. Such transitions are used as an input to our symbolic model checker APALACHE.

Since TLA+ and SMT have different levels of expressiveness, we developed a set of reduction rules from TLA+ to SMT, and implemented them in APALACHE. Instead of enumerating every reachable state like the existing model checker TLC, APALACHE transforms a specification together with desired properties to a set of first-order logical constraints. Then, an SMT solver, e.g., Z3, is used for finding a model (e.g., finding a bug) or proving unsatisfiability (e.g., proving that a property holds). We compared APALACHE with TLC over several benchmarks that stem from distributed algorithms. Our experiments showed that APALACHE is faster than TLC in verifying inductive invariants and finding a counterexample. However, TLC is more efficient than APALACHE in checking safety properties.

APALACHE works over finite structures. Whenever an algorithm is parameterized or parametric, we cannot verify all of its instances with APALACHE. For example, the Chandra and Toueg failure detector is parameterized in the number of processes and parametric in time bounds. Hence, APALACHE can check only its instances with fixed parameters. Thus, we developed additional techniques to verify all instances of the failure detector.

We defined and formalized the class of symmetric point-to-point algorithms that rely on point-to-point communication channels. The Chandra and Toueg failure detector is an
instance of this class. We proved that symmetric point-to-point algorithms enjoy the
cutoff property. These results allow us to verify a symmetric point-to-point algorithm by
checking only instances with two processes.

We also investigated verification of distributed algorithms under partial synchrony. Partial
synchrony lies between synchrony and asynchrony, and guarantee liveness properties by
assuming the existence of bounds $\Delta$ on message delay, and $\Phi$ on the relative speed of
processes after some time point. Our work resulted in a case study for verification of the
Chandra and Toueg failure detector under partial synchrony. We verified safety in case of
fixed parameters by running the tools TLC, APALACHE, and FAST. To cope with
cases of arbitrary bounds $\Delta$ and $\Phi$, we reduced liveness properties to safety properties,
and proved inductive invariants with desired properties in the SMT-based interactive
theorem prover Ivy. While our specifications are not in the decidable theories supported
in Ivy, our experiments showed that Ivy needs little to no additional user assistance to
prove our inductive invariants.

We see several directions for possible future work related to these mentioned topics.

**Symbolic model checking.** We believe that the work in Chapter 3 is only the first
step towards developing an efficient symbolic model checker for $\text{TLA}^+$. Indeed, many
reduction rules can be optimized for fragments of $\text{TLA}^+$. For instance, we could write
more efficient rules for functions with linearly ordered domains such as integers, or rules
for comparing set cardinalities to integers. More importantly, our framework opens the
doors for applying more advanced techniques such as abstraction \cite{CGJ+03, BMMR01}
and reduction \cite{Lip75, CL98}. Reductions were shown to be efficient for a special class
of fault-tolerant distributed algorithms by \cite{KLVW17b}. We are going to explore similar
techniques, in order to check complex $\text{TLA}^+$ specifications of Raft \cite{Ong14}, Disk
Paxos \cite{GL03}, and Egalitarian Paxos by \cite{MAK13}.

**Cutoffs for symmetric point-to-point algorithms.** Our cutoff results in Chapter 6
are for a property with one or two universal quantifiers over process index variables. We
conjecture that given a correctness property with $k$ universal quantifiers over process
index variables, checking $k$ small instances whose size is less than or equal to $k$ is sufficient
to reason about the correctness of all instances.

**Verification of partially synchronous algorithms.** While our specification in Chapter 7
describes executions of the Chandra and Toueg failure detector, we conjecture that
many time constraints on network behaviors, correct processes, and failures in our
inductive invariants can be reused to prove other algorithms under partial synchrony. We
also conjecture that correctness of other partially synchronous algorithms may be proven
by following the presented methodology. For future work, we would like to extend the
above results for cases where GST is arbitrary. It is also interesting to investigate how to
express discrete partial synchrony in timed automata \cite{AD94}, e.g., UPPAAL \cite{LPY97}.  

\newpage

174
List of Figures

1.1 The two-phase commit protocol in TLA+ as specified in [GL06]. (We have only changed the indentation and comments to save some space.) ........................... 4

1.2 The main phases of APALACHE. ...................................................... 14

2.1 A simple producer-consumer ............................................................. 22

2.2 The Next operator of producer-consumer in \(\alpha\)-TLA+ ....................... 24

2.3 A distributed maximum computation in a ring of three processes in TLA+ 33

2.4 SMT formulas that are constructed when checking the executions up to length 4: using the action Next (left), and using symbolic transitions (right). The gray formulas are excluded from the SMT context during the exploration. 35

3.1 The basic workflow of APALACHE. The dashed area shows the preprocessing phases. ........................................................................................................ 39

3.2 Examples of arenas for data structures in KerA+. The leaf cells are equal to the following constants: \(c_1 = c_{21} = 1, c_2 = c_{22} = 2, c_3 = c_{23} = 3, c_{11} = a, c_{12} = 3,\) and \(c_{13} = 0\) ................................................................. 46

3.3 An arena constructed for the set comprehension \(\{x \div 3 : x \in \{2, 3, 4\}\}\). Every cell \(c_i\) has value \(i\) for \(0 \leq i \leq 4\). Cell \(c_S\) encodes the set \(\{2, 3, 4\}\), and cell \(c_T\) encodes the result of the set comprehension. ........................................ 50

3.4 Picking a set among two set cells \(c_1\) and \(c_2\), pointing to \(c_{11}, c_{12}, c_{13}\) and \(c_{21}, c_{22}, c_{23}\), \(c_{1}^3\) respectively. The result \(c_{pick}\) points to \(c_{1}^{pick}, c_{2}^{pick}, c_{3}^{pick}\), which are picked from three sequences of two cells (in red boxes). .................................................. 52

3.5 (a) The arena constructed for the tuple \(\langle 1, \text{true}, "abc" \rangle\), assuming that the expressions 1, true, and "abc" were rewritten into cells \(c_1, c_2,\) and \(c_3\). (b) The arena constructed for the record \([a \mapsto 1, b \mapsto \text{true}, c \mapsto "abc"\] ), assuming that the expressions 1, true, and "abc" were rewritten into cells \(c_4, c_5,\) and \(c_6\). .................................................. 53

6.1 Given execution in \(G_3\), construct an execution in \(G_2\) by index projection. .................................................. 128

6.2 Construct an execution in \(G_3\) based on a given execution in \(G_2\) ............................... 129

7.1 The message buffer after increasing message ages in case of \(\text{buf} = 6\) .............. 161

175
7.2 The increase of message ages with the abstraction of old messages. In the case (a), we have $\Delta = 2$, $buf = 6$, and $buf' = 4$. In the case (b), we have $\Delta = 2$, $buf = 5$, and $buf' = 6$.

7.3 Encoding the increase of message ages with a counter $buf$, and the abstraction of old messages.

7.4 Sending a new message in TLA$^+$ in case of $\Delta > 0$.

7.5 The Next predicate for the next-state relation in TLA$^+$.

7.6 The RSched predicate for scheduling the Receiver in TLA$^+$.

7.7 Sending a new message in FAST in case of $\Delta > 0$. 

162

162

166

166

167

169
### List of Tables

1.1 General-purpose specification languages and support tools ........................................ 9

2.1 The definition of Sub($\phi$) ................................................................. 25
2.2 The constraints on frozen$_\phi$ ............................................................... 25
2.3 The definition of boolForm($\phi$) ........................................................... 27
2.4 The definition of $\delta_\phi$ ................................................................. 29
2.5 The definition of slice($\phi, S$) ............................................................. 31
2.6 Experimental results. The meaning of columns is as follows: “#expr” is the number of expressions, “#assign” is the size of the assignment strategy (number of assignments), “#trans” is the number of constructed symbolic transitions. ........................................................................... 34
2.7 TLA$^+$ expressions that may be rejected by either TLC or the presented technique .......................... 35

3.1 The language KERA$^+$. We highlight the expressions that do not have counterparts in pure TLA$^+$. ........................................................................................................ 42
3.2 The list of TLA$^+$ benchmarks ...................................................................... 61
3.3 The experiments on checking inductive invariants with TLC and APALACHE that are violated. ............................................................................................................. 62
3.4 Checking candidates for inductive invariants with TLC and APALACHE that are violated. ............................................................................................................. 62
3.5 The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are satisfied. ........................................................................................................ 63
3.6 The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are violated. ........................................................................................................ 64

4.1 The values of sum and count on $e_{\text{before}}$ and $e_{\text{after}}$ ........................................ 74

5.1 The experiments on checking Integrity, Validity, Total Order, and Termination with TLC and APALACHE. In this case, these properties are satisfied. ...................................................... 100
5.2 The experiments on checking inductive invariants IIInv, VInv, and TOInv. .................. 100

6.1 Checking the failure detector [CT96] in the synchronous case ................................. 154

177
7.1 Showing Strong Accuracy for fixed parameters ........................................ 168
7.2 Violating Strong Accuracy for fixed parameters ...................................... 168
7.3 Proving inductive invariants with TLC and APALACHE ............................ 168
7.4 Proving inductive invariants with IVy for arbitrary $\Delta$ and $\Phi$ .................. 171
List of Algorithms

1.1 The eventually perfect failure detector algorithm in [CT96] .......................... 6
5.1 Every process \( p \) in Skeen’s protocol follows the below code [GLC19] .......... 98
1.1 The eventually perfect failure detector algorithm in [CT96] .......................... 113
7.1 Adding a fresh message in IVy ......................................................... 170
Bibliography


Jure Kukovec and Igor Konnov. Type inference for TLA⁺ in APALACHE. In *TLA⁺ Community Meeting*, 2020.


[LS79] Butler Lampson and Howard E. Sturgis. Crash recovery in a distributed data storage system. This unpublished paper was widely circulated in samizdat., 1979.


[Mic] Microsoft and HP. The TLA+ Toolbox. URL: [github.com/tlaplus](github.com/tlaplus), accessed: July 2021].


