TU WIEN Informatics

SMT-driven techniques for verifying distributed systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Jure Kukovec , mag. mat.
Matrikelnummer 01652339

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Igor Konnov, PhD.
Zweitbetreuung: Privatdoz. Dipl.-Ing. Dr.techn. Josef Widder

Diese Dissertation haben begutachtet:

Constantin Enea	Viktor Kunčak

Wien, 21. März 2024

Jure Kukovec

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

TU WIEN Informatics

SMT-driven techniques for verifying distributed systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Jure Kukovec , mag. mat.
Registration Number 01652339

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Igor Konnov, PhD.
Second advisor: Privatdoz. Dipl.-Ing. Dr.techn. Josef Widder

The dissertation has been reviewed by:

Constantin Enea	Viktor Kunčak

Vienna, 21st March, 2024

Jure Kukovec

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

Erklärung zur Verfassung der Arbeit

Jure Kukovec , mag. mat.

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, 21. März 2024

Jure Kukovec

Abstract

Distributed systems, that is, networks of real or virtual components that collaborate in solving an algorithmic task underpin many of our modern technologies. The correctness of these systems is often critical; unpredictable system behavior can, in the worst cases, result in major financial loss, confidential data breaches, or even physical harm. Because of this, formal verification of these systems has been an active area of research for many decades. With the advent of cloud services and cryptocurrencies, there has also been an increase in demand, for newer, better, and more robust verification techniques.

The primary contribution of this thesis is a set of techniques for designing a symbolic model checker for TLA$^+$, with which one can verify properties of distributed algorithms. We focus particularly on specifications of distributed algorithms, as the use of TLA$^+$ has recently been seeing rapid growth in the development of distributed systems. Specifically, we:

1. Formalize the notion of a symbolic transition, an equivalence class of transitions in the state-space defined by an algorithm specification. This is both a necessary prerequisite for symbolic verification, as well as a finite abstraction, reducing a potentially infinite number of real transitions, to a finite family of symbolic ones.

2. Define an encoding of constructs in the kernel of TLA$^+$ to first-order logic, suitable for SMT solvers. This allows us to symbolically encode a bounded execution of the algorithm specification, and the properties we wish to verify, as an SMT formula, which can be passed to an off-the-shelf SMT solver.

3. Design a type system for TLA$^+$, as well as a technique for automatic type inference. Types are both necessary, to bridge the gap between conventionally untyped TLA$^+$ and typed SMT, and helpful, because algorithm designers are typically familiar with types and their uses in programming languages, and many errors, especially in the incremental design of an algorithm, can already be caught by type analysis.

Contents

CHAPTER 1

Introduction

1.1 Distributed systems

The term *distributed systems* refers to a broad category of systems, where tasks are performed not by a single unit or device, but collectively by multiple (real or virtual) components. While the original meaning was restricted to physically separated computational units, the modern interpretation, brought about by new technologies, for example multiprocessor hardware, has been extended to include such systems as well. Distributed systems are incredibly common. Modern telecommunications (phone, emails, etc.) might be the most obvious, but there are many examples of technologies enabled by less visible distributed systems.

Example: Cryptocurrencies. In 2008, the now famous cryptocurrency Bitcoin [Nak] was invented, and eventually became a hosehold name, followed by others, such as Ethereum's ether, or Cosmos' atoms. Perhaps more interesting than the currency itself was the technology behind it: blockchain [NGHS17]. To understand the significance of this technology, we must first understand real currency, and the holding and transfer of money. Suppose Alice and Bob both own accounts at the same bank (for simplicity). In order for Alice to transfer Bob some money, she must request a transfer from the bank. The bank then reviews the transaction, approves or denies it and updates the balances of both parties accordingly. Importantly, the bank is the final arbiter for all information disputes; Alice and Bob, and an arbitrary number of external witnesses, may all agree that a transaction was possible (i.e. did not exceed the balance of Alice's account), that it happened, and that the new balances have changed, but as long as that the bank disagrees, the transaction effectively never happened. Worse still, in the hypothetical case that the bank's database is compromised (or just faulty), neither party has the technological means to dispute fictitious transactions. Cryptocurrencies, on the other hand, operate without such central authorities. Underpinning blockchain is the

problem of *distributed consensus* [PSL80] [Lyn96, Chapter 7, 12], in other words: actors in the blockchain system collectively decide truth and history. When an actor proposes a transaction, other actors are rewarded for validating it (e.g. proof-of-work [Nak, Section 4]). Theoretically, unless a single actor owns the majority of the processing power of all actors combined, the collectively computed state and history are reliable, which makes blockchain highly resilient to both faults and malicious agents (see Byzantine fault tolerance [Lyn96, Chapter 2.2]).

◁

Example: SETI. Other famous examples include a program by the Search for Extraterrestrial Life Institute (SETI), called SETI@home [ACK+02]. Their task, analyzing vast amounts of radio telescope data, was very computationally expensive. Researchers appealed to the public for help: individuals could download a program to run on their home computers, presumably overnight or when idling, which would perform some of the computational work and send it back to the researchers. According to their website [set], the initiative ran for over 20 years, logging millions of years of computing time. They claim the cumulative network of computers could at its peak be considered as one of the most powerful supercomputers on the planet. The success of this initiative spurred several other public-assistance driven programs, like Folding@ home [P+10] or climateprediction.net [SAF+04]. ◁

Example: Torrents. Its connection to piracy notwithstanding, torrenting, as a technology, is an interesting method of disseminating data among a network of participants. Assume we know how to send a file from one process (computer) to another via a direct (peer-to-peer) connection. Torrenting works in the following way: An initial process A (seed) lists a file it owns as available for dissemination. Another process B (leech) establishes a peer-to-peer (P2P) connection to process A. However, when process B finishes copying the file from process A, process B becomes a seed as well. Next time, when a process C intends to copy that same file, instead of copying the entire file from A, it establishes two connections, to both A and B, and copies parts of the file from each. As more processes act as seeds, the transfer speed and robustness improve, as a single faulty or slow P2P connection becomes less of a bottleneck, due to redundancy. While this is a simplification, as there are more components in a real torrenting ecosystem (indexers, trackers), as well as dynamism in the network topology, as processes may arbitrarily abandon their roles as seeders or leechers, at its core, torrenting is one of the more elegant protocols, implementing data dissemination in a distributed network from P2P connections. ◁

Note that we list the above examples merely as illustration of the fact that distributed systems are widespread in everyday life, not necessarily as systems this work intends to reason about.

What makes the distributed setting different from the single-component setting is communication; while the execution and correctness of a task/algorithm is important in both

scenarios, the distributed setting must additionally account for information exchange: at certain points in time, a component must share the results of its computation with other components.

Nancy Lynch categorizes distributed systems in her textbook The Distributed Algorithms [Lyn96] based on three facets: the communication model, the synchrony model, and the fault model. While not part of that characterization (though certainly considered in the book), we consider another facet: network topology. Network topology dictates which components are connected. Typically, this is represented by a directed graph, in which vertices are components and edges represent the ability of one component to send information to another. For example, one of the simplest families of distributed system are rings: systems where the network is represented by a connected graph, in which every vertex has exactly one outgoing and one incoming edge. Properties of distributed algorithms over rings have been extensively studied, see [Lyn96, Chapter 3] or [BJK+15] for more details. Communication models, on the other hand, specify how processes exchange information. Examples of different models include peer-to-peer messaging, in which a message is sent individually to each recipient (if multiple), broadcast messaging, in which a message is sent once, and received by all of its intended recipients, and shared-memory based communication (e.g. in a multiprocessor setting), where actors read from, and write to, overlapping regions of memory. Other characterizations include instant- versus queue-based messaging and more. Synchrony models specify the granularity of actions: in synchronous systems, time is abstractly discretized into steps and each component performs one sub-task and sends messages (which are also received) in each step, while in asynchronous systems, time is abstractly continuous and individual components perform their tasks not in pre-specified intervals, but whenever they have the ability to do so, for example whenever a message, with delivery time, is received (however, there is no upper bound on delivery time). Partially-synchronous systems lie somewhere between these two extremes, for example when upper bounds on delivery time exist, but are not known, or when the system starts out behaving asynchronously, but becomes synchronous after some unknown period of time.

Lastly, distributed systems can sometimes have an advantage over single-component systems, in the sense that they may be able to tolerate faults. In a single-component setting, if said component fails, e.g. for hardware reasons, this is typically unrecoverable. However, there exists a family of distributed systems, called *fault-tolerant systems* [Avi76, KK20], for which a limited number of faults, either at the component- or at the communication-level is manageable, in the sense that the correct parts of the system may collectively compute the correct result, in spite of the faults present. Fault models range from more simple, where a component completely ceases to function (crash faults) or becomes unable to send messages to a particular set of other components (unreliable communication), to more complex, in which a component may act maliciously and send arbitrary messages to other components (byzantine faults). Fault-tolerant distributed systems play an important role in ensuring stability, often at the cost of some minor redundancy, for example in modern cloud-based data storage [ZXHW10].

Famous examples, and the likes of which we analyse in later parts of this work, include Paxos [L+01] (and many of its derivatives) and Raft [Ong14], crash-fault-tolerant distributed consensus algorithms, or Tendermint [BKM18], a byzantine-fault-tolerant distributed consensus algorithm for blockchains.

Example: Paxos. Paxos, first introduced by Leslie Lamport in [Lam98] as a story about a fictional island and its legislators and later summarized in a more traditional paper-format in [L+01], is an algorithm, that tackles the problem of *distributed consensus*. Specifically, given a set of processes with the ability to propose values, it describes the steps the processes must take to agree on a single value, such that a) that value was proposed by at least one of the processes, b) the agreed-upon value is the same for all processes, and c) no process mistakenly assumes consensus, where none exists. The requirements on the processes are quite lax: processes are asynchronous, meaning that time intervals between two actions of a process are arbitrary, and may exhibit crash faults (and recovery). The presence of crashes and restarts means that, barring additional constraints, the algorithm does not guarantee termination. The inter-process communication is similarly permissive: message delivery times are arbitrary, and any message sent may arrive more than once, or never at all (but remains unaltered, if it does arrive). A high-level outline of the algorithm is as follows: Participating processes are marked as proposers or acceptors (or both), the first deciding which values are taken into consideration for acceptance, and the latter deciding which proposed value to actually accept. One proposer broadcasts its intent to select a value (but not the value itself) to a majority of acceptors, i.e., sends a *prepare request*. Acceptors return a message acknowledging the receipt of a prepare request, and include any accepted value proposals (since the message may have arrived late, due to communication delay). If a proposer receives acknowledgments from a majority of acceptors, it either selects the most appropriate value (details on how this is selected are omitted here), from value proposals included in the acknowledgment, if there are any, or an arbitrary value otherwise. This step allows proposers to catch up with information about value proposals made by other processes. The selected value is then sent to acceptors as a part of an *accept request*. Any acceptor that receives an accept requests accepts the proposal, except in the case that it has already responded to a prepare request with greater precedence (details omitted). In addition to the above, Paxos describes a third category of processes, learners, responsible for detecting that a majority of acceptors have agreed to accept the same proposal. For more details, see [L+01, Lam98]. ◁

1.2 Challenges

In this section, we present the central challenges addressed by this work. Before we can formulate them, however, we must first introduce the notion of *specifications*.

Specifications. A *specification* of a (distributed) system or algorithm, is the formalization of said system or algorithm and its logical properties in some *specification language*

(for example, TLA$^+$ [Lam94]). Specification languages, unlike natural languages, have explicit and unambiguous semantics, and typically borrow notation from mathematics or programming languages. Formal semantics and lack of ambiguity then allows for the existence of tools, with which these specifications can be automatically verified. For example, the natural-language expression "... receives a message from at least one correct process", which one is likely to find when reading about fault-tolerant algorithms, could be represented as "Cardinality(messages) $\geq f + 1$" (where f is an upper bound on the number of faulty processes). The work presented in this thesis focuses primarily on the specification language TLA$^+$.

General verification challenge. Given a specification of a distributed system, or parts thereof, and its behavior in a suitable formalism, for example TLA$^+$, and a property of the system described in the same formalism, we wish to verify whether the property holds true of the system and/or its behavior.

Unfortunately, the general verification challenge is far too broad. Specifically, many system properties encode within, or reduce to, other known undecidable problems. For example, [KKW18] shows that deciding a form of reachability in the system's state space in some cases reduces to solving the halting problem for two-counter machines, which is known to be undecidable [Min67, BJK$^+$15].

Knowing this, we cannot hope to tackle the general verification challenge. However, we can always restrict the general setting in some reasonable way, and attempt the sub-challenges arising from such restrictions. In this thesis, we address a sub-challenge of the general verification challenge, which is more realistic in scope:

Bounded model checking of TLA$^+$. In this restriction of the general verification problem, we restrict both the formalism and the class of properties. Specifically, we are interested in systems and properties which are described in the specification language TLA$^+$. Moreover, we are only interested in properties that hold true of a bounded execution of the system. Given these inputs, our goal is to encode the bounded system execution, i.e. the set of states reachable in a bounded number of steps, and the property to be verified in a fragment of first-order logic, supported by satisfiability-modulo-theory (SMT) solvers. Then, we can use 3$^{\text{rd}}$ party off-the-shelf solvers to obtain models, or unsatisifiability proofs, of our encoding and recover either a proof that the input property holds true of the system, or a counterexample trace.

1.3 Contributions

In this work, we address Bounded SMT-based model checking of TLA$^+$ in the following way:

We design a symbolic-transition decomposition of TLA$^+$ transition-predicates in Chapter 6 . In Chapter 7, we characterize the language-kernel of TLA$^+$, KERA$^+$, and present

KeRA$^+$ operational semantics as sound reduction rules, which turn KeRA$^+$ formulas into equisatisfiable SMT constraints. Lastly, we design a type-system for TLA$^+$ and implement type-inference for TLA$^+$ specifications in Chapter 8. We implement all of the above techniques in a symbolic model checker, Apalache.

1.4 Applications

The TLA$^+$ model-checker Apalache [KKT19a], which I am a main co-author of, is the culmination of all of the techniques described in the thesis, as well as significant engineering and UX effort. This thesis mentions, but omits details of, various forms of preprocessing or model reconstruction, for the purpose of simplifying presentation or drawing attention to the more theoretically interesting parts of the work; in practice, the additional work is equally pivotal in guaranteeing both base functionality, as well as performance. It is worth emphasizing that Apalache performs several analyses and preprocessing phases, including desugaring, normalization, renaming, transformation to KeRA$^+$, inlining, and more.

Having started as an academic prototype, it was officially adopted by Informal Systems in 2020, and has since seen use both internally, as a formal verification tool for blockchain-related systems and protocols, as well as externally, as a support tool in security audits. In particular, it has been used to verify parts of the Tendermint [Buc16], and LightClient protocols [ten] at Informal Systems, and to generate test suites derived from specification counterexamples, as part of Informal's auditing and consultancy work for its business partners. This speaks to the usefulness of the techniques presented in this work, as well as the usefulness of symbolic model checking for TLA$^+$ as a whole. Apalache is free, open-source software, and will remain as such in the future. A list of Apalache applications can be found on the Apalache web page [Sys20]

1.5 Publications

Some parts of this work are based on existing publications. Some publications are not extended in this work, but describe work done in the domain of distributed algorithms or formal verification :

- Conference paper at ABZ'18 [KTK18], which was later extend to a journal version in the journal Science of Computer Programming 187 [KTK20]: Extended in Chapter 6.
- Journal paper at OOPSLA'19 [KKT19b]: Extended in Chapter 7.
- Conference paper at CAV'19 [BBC$^+$19]: This work has been done during the author's internship at Amazon AWS. It demonstrates the use of SMT-based techniques in industrial-scale applications, developed for use by Amazon Web Services, to

reason about network reachability, as a means to track down and remove virtual-network access vulnerabilities.

- Conference paper at CONCUR'18 [KKW18]: Presents a comprehensive characterization of reachability for families of threshold automata, a formalism used in the verification of threshold-based distributed algorithms, based on the shape of their guards, the presence of decrements and constraints on loop-behavior.

CHAPTER 2

Thesis roadmap

In this thesis, we present a comprehensive approach to symbolic TLA$^+$ model checking, and the various components required to facilitate it. Despite TLA$^+$ already being used by major players in the industry (AWS, Microsoft, Oracle), as well as startups (Informal Systems) , and steadily gaining popularity, tooling availability remains one of the largest pain-points of the field, as explicit-state modeling approaches, despite continuous engineering efforts, face an unsurmountable wall, in the form of the state-space explosion problem, by their very nature. Chapter 3 gives a brief overview of TLA$^+$, intended for readers unfamiliar with the language, and includes a discussion of the features of TLA$^+$, which make designing tools for formal reasoning about it difficult.

We believe it is vital for the wider adoption of the language, going forward, to design and implement different approaches to model checking. To this end, the thesis lays the groundwork for a symbolic model checking approach, and addresses the challenge outlined in Section 1.2. Chapter 4 provides an introduction to explicit-state and symbolic approaches to model checking.

In Chapter 5, we give an overview of the techniques, tools, and specification languages used in the study of distributed systems, showcase efforts to introduce type systems for other untyped languages, and briefly introduce the ByMC model checker, used in the study of threshold-based distributed algorithms.

Symbolic transition decomposition. The first step towards symbolic model checking is transition decomposition. In Chapter 6, we formalize the notion of an assignment strategy, the existence of which encodes the ability of a model checker to constructively generate a symbolic encoding of a successor state, from a symbolic encoding of the current state. This is important, because it often keeps the reachable state space finite, or at least finitely representable. It also allows for an assessment of type-compliance. Using assignment strategies, we then give a formalization of slices and symbolic transitions,

to be used as a starting point for model-checking. We solve the assignment problem by means of an SMT solver, and the decomposition by means of syntactic processing, and our approach is provably sound.

This process serves two purposes: firstly, the search for assignment strategies is a form of static analysis, able to discover parts of the specification where behavior is under-specified, from the perspective of a model checker. For example, this helps us eliminate scenarios where the state-space is theoretically well-defined, e.g. when the transition relation specifies only cyclic constrains, but the input would result in either obvious non-termination (explicit-state exploration), or the inability to encode all successor states (symbolic approach). Secondly, slicing is a form of optimization, reducing the burden on SMT solvers, by reducing a monolithic formula into multiple smaller ones, that can be independently analyzed.

Encoding TLA$^+$ in SMT. To perform bounded model checking, we must be able to take a (symbolic decomposition of a) specification, and translate one execution step of the system described by it as a first-order logic formula. In Chapter 7, we introduce the kernel of TLA$^+$, called KerA$^+$. Then we introduce sound rewriting rules, one for every kernel operator, that iteratively simplify a TLA$^+$ formula, producing SMT constraints and an auxiliary structure, called an arena, in the process. If all assumptions outlined in Chapter 7 are met, the final result of transforming a specification is an SMT formula that is equisatisfiable to the specification (i.e. the SMT formula is equivalent to *true* in FOL iff the specification is equivalent to TRUE in TLA$^+$).

This forms the basis of a symbolic bounded model checker: to encode whether a k-step execution satisfies an invariant, we perform the above translation once, but parameterize it by the SMT representation of the current/next state variables. Then, we instantiate it with k pairs of SMT variables, for $(x_0, x_1), (x_1, x_2), (x_2, x_3)$, and so on. The SMT solver is asked to then solve the formula encoding invariant violation, so an UNSAT result is equivalent to saying that every k-step execution satisfies the invariant, whereas a SAT result, and the model it comes with, represents an execution which violates the invariant.

Note that in practice, instead of encoding a k-step execution directly, we iteratively check one step at a time, up to k times, attempting to find an invariant violation after every step.

Type inference for TLA$^+$. The translation to SMT, outlined in Chapter 7, assumes that every expression in the specification can be assigned a type in our type system. To keep the chapter focused on just the encoding, we glossed over the fact that, in practice, introducing manual annotation is often cumbersome to users. Therefore we describe a fully automatic type inference process in Chapter 8. To do this, we need to slightly expand the type system, but it remains fully backwards-compatible with the type system referenced in the rewriting rules.

In our experience, it is often the case that type errors hide correctness bugs. It is therefore valuable, especially in the specification-writing stage, to have some type analysis utility, either in the form of a type-checker or type-inferer, which can both easily be spun out of the model checker into standalone tools.

CHAPTER 3

A primer on TLA$^+$

The Temporal Logic of Actions (TLA) is a specification language originally introduced by Lamport [EGL92, Lam94]. It was intended to be used to reason about software, or more broadly, about systems of interconnected components, that change over time, building upon Pnueli's temporal logic [Pnu77] and its associated temporal operators. It was later extended to TLA$^+$ [Lam99, Mer08a], with support for first-order logic (FOL) [Bar77], Zermelo-Fränkel set theory [Cie97] and modularity, at the language level. TLA$^+$ does not fix a model of computation, and thus it found applications in the design of diverse concurrent and distributed systems, e.g., see [GL03a, NRZ$^+$15, Ong14, MAK13, AMW16].

3.1 The fundamentals of TLA$^+$

In this section, we give a brief introduction to TLA$^+$, that focuses on the language itself, rather than describing how to use it to write "good" specifications. A more complete description of the language can be found in [Lam02], [Way18] or [KKKF20].

Programs generally describe every minutia of a system; the primary emphasis is on establishing *how* a computer performas a change. For example, in a list-sorting algorithm, there are many ways of transforming an unsorted list into a sorted one. To name a few, bubble sort, quicksort and merge sort [KK73] are all distinct ways of sorting a list, with distinct runtime complexities and space requirements. However, on a more abstract level, applying any of the above algorithms to an arbitrary list results in the same outcome – a sorted list. This is where specifications, such as those written in TLA$^+$ differ from programs; instead of describing the process of sorting a list itself, which is a generally well studied problem, it is instead often sufficient to abstract the process, by describing a transition from an initial state, one where the list is arbitrarily ordered, to a successor state, in which the list is sorted. TLA$^+$ allows users to do precisely that. By design,

the language uses standard mathematical notation and terminology. Figure 3.1 depicts several ways of specifying a list-sorting algorithm.

The exact state-space model underpinning TLA$^+$ is as follows: assume a universe of values $\mathcal{U}$. For now, the exact contents of $\mathcal{U}$ are unimportant. Then, a k-variable state space is simply the cross product $\mathcal{U}^k$, and a state is a k-tuple. A specification, for the purposes of this section, defines a pair (I, T), where $I \subseteq \mathcal{U}^k$ is the set of initial states, and $T \subseteq \mathcal{U}^k \times \mathcal{U}^k$ is a transition relation. Note that, in theory, I is not required to be finite, nor is T required to constrain finitely many successors for every state reachable from I. In practice, however, tools often reject infinite-state specifications as, for example in the model checker TLC [YML99], state enumeration becomes impossible if the number of states is not finite. We later show that our symbolic approach can also work in cases where the state space is potentially infinite (see Section 4.2).

There are several well-studied formalisms available for reasoning about system models, for instance Kripke structures or, more broadly, (labeled) transition systems [CHVB18]. Unfortunately, they don't always capture the state space a TLA$^+$ specification defines. For instance [CHVB18, Chapter 3.5.1] defines Kripke structures to be finite, but the state space defined by a specification does not have to be. Similarly, transition systems, as defined in [Kel76], are uninitialized, that is, they define a transition relation over pairs of states, but don't constrain the initial states of the system, unlike a specification. In the case where the state space is finite, one can, however, view the state space as a Kripke structure, where the set of atomic propositions (AP) contains all the invariants defined in the specification. In general, TLA$^+$ specifies infinite-state transition systems, in the above sense.

We will use list-sorting, a specification of which can be seen in Figure 3.1, as a running example to illustrate the use of TLA$^+$. The first thing of note, in relation to the previously mentioned modularity of TLA$^+$, is that specifications may build upon one another. Much like a header inclusion in C, or an import statement in Java, line 1 in our example specification indicates that this specification borrows operators from well-known integer arithmetic; the integer operator symbols, such as $+$, hold their standard meanings, when used with integer arguments. The latter part of the above sentence is important, because both $1+1$ and $1+$"*abc*" are legal TLA$^+$ expressions and, more specifically, both elements of $\mathcal{U}$. We know $1+1=2$, but the value of $1+$"*abc*" is not specified. Specifically, it is not necessarily the case that $1+$"*abc*" $\neq 2$.

The following line 2, uses the keyword CONSTANT to declare constants in the specification. Unlike in programming languages, constants are abstract at the specification level. Typically, they are used as a way to parameterize the specification. In our example, the specification admits two constants, *listElements* and *listSize*, representing the possible contents of the lists being sorted and the size of the lists respectively. These constants are used, because the process of sorting a list, and the property of a list being sorted, are conceptually the same for lists over any linearly ordered domain and of any length. While, in TLA$^+$, the "$\leq$" operator is only defined for integers, it is often used to model

Figure 3.1: A Specification of a List-Sorting Algorithm in TLA$^+$

```
1  EXTENDS Integers
2  CONSTANT listElements, listSize
3  VARIABLE list
4
5  Domain ≜ 1..listSize
6  Lists ≜ [Domain → listElements]
7
8  Init ≜ list ∈ Lists
9
10 IsSorted(a) ≜ ∀ i ∈ 1..(listSize−1): a[i] ≤ a[i+1]
11
12 Next1 ≜ IsSorted(list')
13
14 IsInjective(f) ≜ ∀ x,y ∈ DOMAIN f: x ≠ y ⇒ f[x] ≠ f[y]
15 Permutations ≜ { pi ∈ [ Domain → Domain ]: IsInjective(pi) }
16 IsPermutationOf(a, b) ≜ ∃ pi ∈ Permutations: ∀ i ∈ Domain: a[i] = b[pi[i]]
17 Next2 ≜ ∃ sortedList ∈ Lists:
18   ∧ IsSorted(sortedList)
19   ∧ IsPermutationOf(list, sortedList)
20   ∧ list' = sortedList
21
22 Next3 ≜
23   IF Sorted(list)
24   THEN UNCHANGED list
25   ELSE list' ∈ { a ∈ Lists: IsSorted(a) ∧ IsPermutationOf(a, list) }
```

systems where a linear order exists over other domains, such as the lexicographic order on the set of strings, often used in programming.

This allows us to describe the property defining a sorted list, via the comparisons of sequential elements, without explicitly incorporating details that might be pertinent in programming languages, such as the sort of element being compared or the exact length of iteration.

Next, and truly at the heart of TLA$^+$, is the declaration of state variables, by the use of the VARIABLE keyword in line 3. State variables define the states of a state-machine that a specification describes; one state is defined by a combination of values held by each state variable. In our particular example, each state is uniquely defined by the value of *list*. An important consideration, when talking about the state-space defined by a specification, is that TLA$^+$ is an untyped language. This means that there exists a state where *list* has the value 7 or "abc", despite the fact that 7 and "abc" are not lists in any intuitive sense. Other parts of the specification (*Init*) will ensure that exploration starts

in a reasonable state, i.e. a state where *list* holds a value that abstractly describes a list, and that it transitions (*Next*) into a state where that remains true (in addition to being sorted). However, at the level of the state-space, these unintuitive states still exist. Each variable declared in a specification may appear in one of two forms: unprimed, for example *list*, and primed, for example *list*′. In the context of the state-space being described, unprimed variables reference values in the current state, whereas primed variables reference values in successor states, relative to the current state. Formulas which reference primed variables can then be thought of as describing transitions in the state space.

Lastly, the specification consists of a series of operators, defined with " $\triangleq$ ", abstractly describing various constructs or properties. For example, we understand *Lists* to represent the set of all lists of size *listSize*, the elements of which belong to *listElements*. Whether this set is finite or not depends on the latter. If we assumed that $listElements = \{0, 1\}$, for example, the set *Lists* would have a size of $2^{listSize}$ (equivalent to the set of all binary words of length *listSize*). On the other hand, if we assumed $listElements = 3$, this construct would be undefined and, conceptually, nonsensical, but it is nevertheless the case that $[Domain \to 3] \in \mathcal{U}$.

Similarly, we understand *IsSorted*(_) to be a unary predicate describing sortedness – TRUE, if the argument is a sorted list and FALSE if it is an unsorted list. Note again, however, that since TLA$^+$ is untyped, there is no requirement that a is a list of any kind; *Sorted*(7) is a syntactically valid TLA$^+$ expression and equals some unspecified value in $\mathcal{U}$.

Certain operators are of special significance to the specification. Each specification will designate an initial-state predicate, commonly named *Init*, or some variant thereof. This operator defines the starting point in state-space exploration. We've mentioned before, in the list-sorting example, that our intuitive understanding of the variable *list* was that of a list-abstraction, and not, say, an integer. The operator *Init* makes this explicit – in the initial state *list* belongs to *Lists*, the set of all functions, the domain of which is $\{1, 2, \ldots, listSize\}$, and the codomain of which is *listElements*. Such functions serve as our abstraction of lists (of a given length, with given contents). For example, in Figure 3.1, one possible execution, given the constants $listElements = \{1, 2, 3\}$ and $listSize = 3$, would begin with the initial value of *list*, for which $list[1] = 3$, $list[2] = 2$, $list[3] = 1$ (and DOMAIN $list = 1 \,..\, 3$), satisfying *Init*.

Similarly, specifications also designate a top-level transition operator, commonly named *Next*, or some variant thereof. This operator, together with the initial-state predicate defines state-space exploration, by providing constraints, which must hold in 1-step reachable states from any given state. In our example, we provide three alternative ways of expressing transitions into states which represent sorted lists.

The first operator, *Next*1, succinctly mandates that the value of *list* in any 1-step reachable state satisfies the *IsSorted*(_) predicate. While we might intuitively believe that this sufficiently describes states representing sorted lists, recall the following particularity

of TLA$^+$: because it is untyped, $IsSorted(7)$ is a syntactically correct, but undefined, formula. Therefore, we cannot say that it is categorically untrue that a state, in which $list = 7$ satisfies the $IsSorted(\_)$ predicate. But even on a more fundamental level, even if we assumed that $list'$ is a list-abstraction (i.e. an element of $Lists$), $Next1$ does not establish the relation between $list'$ and $list$; in this sense, $list'$ may be sorted, but it is not necessarily a rearrangement of the elements of $list$. In fact, any list with $listSize$ identical elements would do as $list'$, which is clearly not the correct outcome of list-sorting, as sorting is expected to preserve list elements. Continuing our example, where $listElements = \{1, 2, 3\}$, $listSize = 3$, and $list[1] = 3$, $list[2] = 2$, $list[3] = 1$ holds in the initial state, $Next1$ can be satisfied by $list'[1] = 4$, $list'[2] = 4$, $list'[3] = 4$, $list'[42] = 9$ (and, e.g. DOMAIN $list' = 1 \,..\, 100$), even though $list'$ is neither a permutation of $list$, nor even an element of $Lists$.

The operator $Next2$, while more verbose, is also more precise: since $Lists$ represents the set of all lists (of a given length, with given contents), it stands to reason that one of its elements ($sortedList$) is a sorted list. The predicate then asserts, that the value of $list'$ equals a list that is a) sorted and b) a permutation of the original list. It must now also be the case that $list'$ is a member of $Lists$. For $listElements = \{1, 2, 3\}$, $listSize = 3$, and $list[1] = 3$, $list[2] = 2$, $list[3] = 1$, this time the only choice of $list'$ that satisfies $Next2$ is one where DOMAIN $list' = 1 \,..\, 3$, $list'[1] = 1$, $list'[2] = 2$, $list'[3] = 3$, which is the expected outcome of sorting $list$.

Finally, the operator $Next3$ has a similar issue to $Next1$; for non-list values of a, it is not known whether $IsSorted(a)$ holds or not. However, in combination with $Init$, which asserts that we start state-space exploration in a state where $list$ is a list, the operator $Next3$ correctly constrains successor states where $list$ is both sorted and a member of $Lists$. Our running example, for the initial state where $listElements = \{1, 2, 3\}$, $listSize = 3$, and $list[1] = 3$, $list[2] = 2$, $list[3] = 1$, mandates the exact same choice of $list'$ as for $Next2$.

While list-sorting is a good starting example, it does not showcase the power of TLA$^+$. In [erc], we showcase a specification of ERC20, an Ethereum technical standard for smart contracts.

3.2 Peculiarities of TLA$^+$

This section disusses some features of TLA$^+$ that are of special interest to us, not because of their expressibility, but because they present a significant hurdle from a tooling or verification perspective. These language features highlight the fact that TLA$^+$ was never designed with machine automation in mind, and this section offers insight on how and why TLA$^+$ is restricted to the fragment we use in practice.

Actions and implicit assignments. One way in which TLA$^+$ differs significantly from many other languages, is its approach to state transitions. For example, in programming languages, variable updates, which define transitions between program states, are typically

indicated via the use of special assignment operators, such as "=" (as opposed to "=="), or "$\leftarrow$". Importantly, even in the case of updating to, for example, a random value from a (finite) collection, this relation is explicit, in the sense that a) the assignment operator is not symmetric, and the single variable being updated is clearly distinguished (typically by being on the left-hand side of the operator), as well as b) the new value (or possible range of values) is constructively given. That is not the case in TLA$^+$. Consider the following TLA$^+$ formulas:

1. $a' = b$
2. $b = a'$
3. $a = b'$
4. $a' = b'$
5. $(a') * (a') = (b') * (b')$
6. LET $F(x) \triangleq x = b$ IN $F(a')$ $\quad(*$ define operator F and apply it to a' $*)$
7. $G(a, b, a', b')$ $\quad(*$ apply operator G to a, b, a', b' $*)$

All of the formulas are actions, that is, predicates evaluated over a pair of states: a current state, determining values for a, b and a successor state, determining the values of a', b'. A (directed) transition exists between the two states, if the action equals to true, under the evaluations of a, b, a', b' defined by the two states. In contrast to the programming-language setting, however, the relation between the two states is generally implicit. This means that a' and b' are not necessarily computable from the values of a and b. Moreover, the "=" operator used in the above formulas is not an assignment operator, and example (1), for instance, does not constitute an assignment (of b to a). In fact, the notion of an assignment does not exist in TLA$^+$ at all, all of the above formulas are merely restrictions on the transitions that may exist in the state space. Moreover, one can write trivial specifications, which have no possible implementations, because they describe a state space without transitions, for example, by constraining $(x') * (x') = -1$.

Even beyond implicit assignments, one can write, for example, $a' = 1 \wedge a' = 2$. While an imperative language might treat $a' = 1; a' = 2$ as two sequential assignments, first writing 1 to a, then overwriting it with 2, ending in a state where a holds the value 2, in TLA$^+$ this conjunction represents a transition to a state, where a has some value v, such that $v = 1$ and $v = 2$ simultaneously. By the transitivity of equality, and the fact that $1 \neq 2$, such a state does not exist.

If the purpose of a specification is merely to describe an algorithm/protocol, this is not usually a problem. However, when it comes to tools, such as model checkers, which need to explore the state space, it is important to be able to determine, exactly or symbolically, successor states from their predecessors. Therefore, in practice, tools require explicit

relations, such as in (1), to guide their state-space exploration (implicitly treating "=" as asymmetric), as well as a way of treating $a' = 1 \land a' = 2$ (e.g. by treating exactly one of the conjuncts as an assignment, and the other as a constraint). The detection and treatment of assignments is the main subject of Chapter 6.

Lack of types. By design, TLA$^+$ is an untyped language. Consequently, it is perfectly valid to construct expressions such as $\{1, "abc"\}$ and apply reasoning, such as $\exists x \in \{1, "abc"\} : x > 0$, which should evaluate to TRUE, since $1 \in \{1, "abc"\}$ and $1 > 0$. However, this poses a significant challenge to automation. The reason for this is that, if tools were to allow constructs such as, for example, non-homogeneous sets like the one above, they have to forgo any kind of internal typed encoding of TLA$^+$, or invent complex data-types to represent such collections. Moreover, the objects might become significantly more complex. Consider the following: $\{\{\langle 1, "abc"\rangle\}, [x \in S \mapsto x + 1]\}$. This is a set, which contains both another set, as well as a function. Attempting to encode the contents of this set would require a data-type akin to Java's Object: very general, but not very useful without manipulation via casting. Even TLC, despite not having a formalized type system, rejects such constructs, as does Apalache. We address types for TLA$^+$ in Chapter 8.

Operators as second-class citizens. While TLA$^+$ does have operators, they are not first-class members of the language. What this means is that TLA$^+$ expressions may reference operators, use them in applications or pass them as arguments to other, higher-order operators, but operators themselves do not constitute TLA$^+$ expressions. Concretely, a TLA$^+$ operator cannot "return" an operator. This makes operators in TLA$^+$ behave more like macros in C with delayed evaluation. From a tool perspective, this is inconvenient, because it essentially creates a hierarchy of objects within a specification, where expressions build operators, which, in turn, build modules. This is further complicated by the fact that LET-IN expressions actually define local operators, and the existence of LAMBDA expressions. Internally, Apalache attempts to reduce operator call-graph complexity by preprocessing optimizations an inlining.

Deterministic choice. Of all the operators in TLA$^+$, perhaps the most idiosyncratic is the CHOOSE operator. Similar to the existential quantifier, which asserts that a property holds true for at least one witness (without specifying the witness directly), the CHOOSE operator produces a witness to the property. However, unlike the existential, if no witness exists, the result is unspecified. For example,

$$\text{CHOOSE } x \in \{1, 2, 3, 4, 5\} : x > 3$$

equals either 4 or 5 (but always the same value, it is not random), while

$$\text{CHOOSE } x \in \{1, 2, 3, 4, 5\} : x < 0$$

may equal any value, even one of $1, ..., 5$, but also ones such as "abc" or $[x \in \{0\} \mapsto \text{TRUE}]$. This operator is commonly referred to as Hilbert's epsilon operator [AZ02]. There are

two problems with this operator: the fact that the operator is meant to be deterministic, i.e. if

$$\{x \in S : P\} = \{x \in T : Q\}$$

then

$$(\text{CHOOSE } x \in S : P) = (\text{CHOOSE } x \in T : Q)\,,$$

and the aforementioned unspecified behavior in the case where no witness exists. Consider, for example, the following expression: $e \triangleq \text{CHOOSE } x \in Int : x > 1 \wedge x < 0$. Since there is no notion of a runtime exception in TLA$^+$, the above expression is valid, however, its value is unspecified. Consider the implications of this on model checking: suppose the specification contains $P(x) \triangleq x > 0 \wedge x < 0$. What is the value of LET $e \triangleq \text{CHOOSE } x \in Int : P(x)$ IN $P(e)$? Since $P(x)$ evaluates to FALSE for all integer values of x, if it is the case that the arbitrary value to which e evaluates is an integer, the above expression is FALSE. If however, e holds some arbitrary non-integer value, e.g. "abc", then $P(\text{"}abc\text{"})$ is equally undefined (and possibly even TRUE). This sort of unpredictability makes it easy for specifications to contain very subtle bugs, that arise from unexpected values produced by CHOOSE expressions. Apalache, which uses an underlying SMT encoding requiring types, CHOOSE $x \in Int : \ldots$ would take the type-hint from the set Int, and encode the type of this expression as an integer type, so the arbitrary value produced is always at least type-correct.

The other issue with CHOOSE, determinism, also warrants discussion. Suppose the specification contains LET $F(t) \triangleq \text{CHOOSE } x \in Int : x > t \wedge x < t$ IN $F(0) = F(1)$. At the TLA$^+$ level, this LET-IN expression should evaluate to TRUE, because CHOOSE is deterministic and

$$\{x \in Int : x > t_1 \wedge x < t_1\} = \{x \in Int : x > t_2 \wedge x < t_2\}$$

for any pair t_1, t_2. We know $e = e$ is tautologically equal to TRUE, regardless of the value of e, as "=" is reflexive. However, any tool will have to construct a representation of the values of both $F(0)$ and $F(1)$ (which skirt static analysis, as they aren't syntactically equal), to be able to perform equality testing.

It is very difficult, especially when dealing with a quantifier-free encoding, to implement this operator in a deterministic way. In Apalache, we choose not to attempt to guarantee a deterministic implementation of the operator. Interestingly, specifications where deterministic CHOOSE matters are rare, and theoretical in nature, for the vast majority of real-life cases, determinism is not relevant.

Rich data structures. From its very inception, TLA$^+$ was designed for expressivity, not efficiency. It has language primitives for defining powersets (SUBSET S), and sets of functions ($[S \to T]$), which can even be combined in arbitrary ways, e.g as

$$[S \to [(\text{SUBSET } T \cup \text{SUBSET } U) \to V]]$$

Effectively none of the common programming languages natively support these data structures (though they obviously allow them to be defined). Having such data structures is a problem (from an efficiency perspective), for a quantifier-free symbolic-analysis approach that deals primarily with properties. Consider:

$$[A \to [B \to C]] = [[D \to E] \to F]$$

If we wanted to symbolically encode these constraints, we would need to reduce them to axiomatic equality; for sets, as well as for the functions which populate them.

CHAPTER 4

Fundamentals of model checking

4.1 A brief introduction to explicit-state model-checking

Concrete systems, unlike their theoretical abstractions, generally have finitely many system states. Moreover, data structures in real systems are often bounded too. For instance, software often deals not with integers, but with bounded integers, the values of which typically belong to $[-2^k, 2^k - 1] \cap \mathbb{Z}$, for some k. Modern CPUs have $k = 64$, while blockchains typically use $k = 256$. Consequently, specifications of this kind of software then have the property that the set I of initial states they describe in the state-space is finite (though possibly enormous), and, for every state s reachable from I by a transition relation T it is the case that s has only finitely many successors. One can then define T^*, the transitive closure of T, for which $T^*(s, s') \iff s = s' \vee \exists s'' \,.\, (T^*(s, s'') \wedge T(s'', s'))$ and the set of I-reachable states, $T^*_I = \{s \mid \exists s_i \in I \,.\, T^*(s_i, s)\}$. It is then not unreasonable to expect that the set T^*_I, that is, the set of all states an execution may potentially visit, is finite too.

The goal of explicit-state model checking is to directly encode the states $s \in I$, and perform an algorithmic traversal, which computes all states in T^*_I. For more details, see [CHVB18, Chapter 5]. Importantly, because state-spaces are often enormous (e.g. adding a single integer counter to a specification of a blockchain component can multiply the size of the state space that needs to be explored by 2^{256}), naive techniques quickly show to be impractical.

We are mostly interested in bounded model-checking [BCCZ99, BCC+03], as it relates to TLA$^+$.

To attempt to model-check a TLA$^+$ specification in practice, several steps are necessary. First, a fragment $\mathcal{U}^e$of $\mathcal{U}$ is chosen, such that all of the values contained within are machine-representable. Typical examples of such values are bounded integers, as well as finite sets and functions. Explicit-state model checking will only succeed, if every

Figure 4.1: A Specification Encoding the Collatz Conjecture

```
1  EXTENDS Integers
2  VARIABLE x
3
4  Successor(n) ≜ IF n % 2 = 0 THEN n ÷ 2 ELSE 3*n + 1
5
6  RECURSIVE kIter(_,_) \* Recursive operators need to be explicitly annotated
7  kIter(a,k) ≜ IF k ≤ 0 THEN a ELSE Successor(kIter(a, k−1))
8
9  ReachesOne(a) ≜ ∃ n ∈ Nat: kIter(a,n) = 1
10
11 Init ≜ x ∈ { n ∈ Nat: ¬ ReachesOne(n) }
12
13 Next ≜ UNCHANGED x
```

component of every reachable state belongs to $\mathcal{U}^e$. Second, all specification constants must be initialized with valid TLA$^+$ expressions.

Computing initial states. The first task is to extract initial states from an initial-state predicate. While it is obviously true, at the logic level, that there is an equivalence between a set of states S and a state predicate $P(\_)$, such that P is true of s iff $s \in S$, in practice, it is not always trivial to determine the shape of such states. For instance, one can encode the Collatz conjecture [Gar81], a notorious open problem in mathematics, about whether a defined iteration of a positive integer function eventually terminates with 1, for any initial value, in TLA$^+$: take a specification with one state variable x and define the initial state predicate $Init \triangleq x \in \{k \in Nat : k \text{ disproves the Collatz conjecture}\}$. This specification is given in full detail in Figure 4.1. However, such an initial state predicate is unusable for any tool attempting explicit state-space exploration, since finding even a single initial state is equivalent to solving the above open problem in mathematics. Therefore, tools like TLC or Apalache generally require that states be explicitly computable from predicates, i.e. that state variables appear as part of an equality or set membership, e.g. $x' = A, x' \in B$.

Computing next states. Once initial states are computed, the process of enumerating all reachable states is conceptually very simple. Performing either breadth-first search or depth-first search, one unexplored state is selected. Then, the transition predicate can be dynamically evaluated: unprimed occurrences of state variables use values from the selected state, whereas primed state variables are used to define successor states. In this sense, once current-state variable values are fixed, the process is identical to the computation of initial states. The exact details, including syntactic restrictions on the order of terms, or the shapes of expressions from which successor-state variable

Algorithm 4.1: Breadth-first-search pseudocode

```
1 queue ← [ state if Init(state) ];
2 while queue not empty do
3 |   state ← queue.pop();
4 |   explore state (e.g. check invariants);
5 |   successors ← [ state' if Next(state,state') ];
6 |   queue ← queue + successors
7 end
```

values are computed are tool-specific. Pseudocode of breadth-first search can be found in Algorithm 4.1. For example, in TLC [Lam02, Chapter 14.2.6] using the breadth-first search approach, a transition predicate is traversed in left-to-right syntax order and the first appearance of a primed variable in this order determines a new state (or family of states) to be added to the exploration queue at the end of the traversal. If the transition predicate is discovered to be FALSE partway through exploration, a state is deemed to have no successors and exploration continues from the next state in the queue. On the other hand, in the absence of errors, if the transition predicate is traversed until the end, one or more successor states is added to the queue. This process continues until no more successor states are found (i.e. the full set of reachable states has been computed).

In theory, if every state-variable can take one of finitely many values, this process is guaranteed to terminate eventually, in the worst case exploring all of the finitely many states. In practice, however, the biggest drawback of this approach is that the number of states that need to be explored, even for relatively simple specifications, can often be quite large. This is known as that state-space explosion problem [Val96] and is unavoidable with explicit state representation. For instance, in the prominent paper by AWS, [New14], TLC was used to explore 31 billion states, which took them "approximately 5 weeks on a single EC2 instance with 16 virtual CPUs, 60 GB RAM and 2 TB of local SSD storage."

That said, there are a number of techniques developed to help lessen the effect of the state-space explosion problem. Some are more theoretical, like symmetry reduction [CEJS98]. With that technique, based on establishing a symmetry-derived equivalence relation over states, one can reason about all states, by visiting only one state per equivalence class, instead of all of them. Others are more practical, like multi-core exploration [HB07], where the computer architecture is leveraged to explore a number of different states in parallel. A more exhaustive list of such methods can be found in [CHVB18, Chapter 1.3.1] or in [Kup17, KLR19a], for TLC-specific methods.

To illustrate how relatively simple specifications may end up describing a large state-space, consider a trivial specification with two state variables x, y, the initial-state predicate $Init \triangleq x = 1 \wedge y = 1$ and the transition predicate that specifies x and

Figure 4.2: A TLA$^+$ specification with N^2 states

```
1 EXTENDS Integers
2 VARIABLE x, y
3 CONSTANT N
4
5 Init ≜ x = 1 ∧ y = 1
6
7 Next ≜ x' ∈ 1 . . . N ∧ y' ∈ 1 . . . N
8
9 Inv ≜ x ∗ y <= N ∗ N
```

Table 4.1: TLC runtimes for the specification in Figure 4.2 with a 6h timeout.

N	runtime (s)
10	6
100	13
1000	**TO**

y taking independent values in $1 \,..\, N$: $Next \triangleq x' \in 1 \,..\, N \wedge y' \in 1 \,..\, N$. Despite its simplicity, this specification has N^2 distinct states. Figure 4.2 contains the full specification. Table 4.1 shows the time needed for TLC to explore all states (and verify the given invariant), for a selected few values of N.

The slowdown in TLC's run-times can be explained by observing the shape of the state-space defined in this specification. We have already established that there are N^2 distinct states, corresponding to the N values each of x and y may *independently* take. However, it is also noteworthy that the transition relation is unrestricted, that is, any state may lead into any other state. In other words, the state space forms a complete graph over N^2 edges (with self-loops), which has N^4 edges. An exhaustive exploration must enumerate not only the states, but the transitions between them, which, in the case of $N = 1000$ is 10^{12} edges. Unsurprisingly, this is incredibly slow.

4.2 A brief introduction to symbolic model-checking and SMT

In practice, the goal of using (explicit-state) model checking is to find counterexamples to the specification or to prove that none exist (provided exhaustive exploration terminates in time); the fact that all states are being explicitly explored is an implementation detail and the exact states are generally of little interest. To this end, we often want to reason not about the state-space per se, but about some quotient space of the state space, commonly defined by some property of interesting states. For example, if the only thing we care about, for each variable, is whether or not it is greater or equal to 3, the above

N^2 explicit states collapse to only 4 equivalence classes of states, regardless of the value of N. This is precisely where the value of a *symbolic* approach lies: if we are able to find a suitable equivalence relation, derived from a constraint in some logic (e.g. linear integer arithmetic - LIA), we can reason about a small set of symbolic states, instead of the much larger set of explicit states. This approach has an added benefit of sometimes being able to capture an infinite state space with finitely many properties, such as slicing $\mathbb{Z}^2$ into 4 quadrants, with the linear inequalities $x \geq 3$ and $y \geq 3$.

The origins of symbolic model checking date back to the invention of binary decision diagrams (BDDs) [CHVB18, Chapter 7], a graph-based representation of Boolean functions. While interesting in their own right, we are more interested in the modern approach that has since evolved from these foundations: satisfiability modulo theories (SMT) [CHVB18, Chapter 11].

To use SMT, we need to be able to a) encode a unifying property of part of the state-space as a formula in some logic (e.g. LIA) and b) compose the formulas (in potentially different logics) with constructs such as Boolean operators, or possibly quantification. Under the hood, SMT solvers split work between specialized theory solvers, for example a LIA solver, and structural satisfiability solvers. The reason is that, no matter what kind of theory the formula p belongs to, it is always impossible to satisfy $p \wedge \neg p$, on a structural basis alone. In an internal loop, the SMT solver first abstracts atoms in the formula with propositional variables and solves the generated Boolean satisfiability problem with a SAT solver. This creates obligations for one or more theory solvers: if atoms a and b in theory T were abstracted as variables v and w, and $v \wedge \neg w$ holds in the model produced by the SAT solver, the theory solver for T must solve $a \wedge \neg b$. If this is satisfiable in T, the loop terminates, otherwise a lemma, a restriction on the assignments found by the SAT solver, is added to the original formula, and the SAT solver is run anew. Eventually, either a satisfying assignment is found and the formula is satisfiable, or all possible propositional solutions are explored unsuccessfully, and the formula is concluded to be unsatisfiable.

The study of SMT solving, and the various techniques and optimizations is of course much deeper than that, an interested reader can find more in [CHVB18, Chapter 11], [DB08, BPF15, MRTB17, TRBB18].

Bounded model-checking. A particular variant of model-checking, to which SMT naturally lends itself is bounded-model checking. The general approach is as follows: First, the length of the execution to be considered, denoted k, is fixed. Then, for each component of the system state v, k SMT variables are introduced, $v_1, \ldots, v_k$, where v_i represents the value of v at the i-th step of the execution. If a transition predicate defines some relation R, between v and v', the values of a component in the current and successor state respectively, the relations $R(v_1, v_2), \ldots, R(v_{k-1}, v_k)$ are introduced as SMT constraints. If one is interested in verifying an invariant I, then $\neg I(v_1) \vee \cdots \vee \neg I(v_k)$ is introduced as well (for all system components v). The full collection of such constraints describes a k-step bounded execution of the system, which violates the invariant. One can

Table 4.2: TLC and Apalache runtimes for the specification in Figure 4.2 with a 6h timeout.

N	TLC runtime (s)	Apalache runtime (s)
10	6	1
100	13	1
1000	**TO**	1

easily recover the counterexample to the invariant from a model of the above constraints. Conversely, unsatisfiability implies that the invariant cannot be violated by any execution of length at most k. Notably, there may exist executions of length more than k, that do violate the invariant, so the technique cannot be used to offer guarantees about arbitrary executions in general, though in some cases, it can be shown that if a counterexample exists, there must exist a counterexample of bounded length, which makes this technique complete, in terms of verifying the desired invariant, see [KVW17]. While this is a general description of bounded model-checking, the difficult part, for particular applications, is designing the correct SMT encodings of the system states, invariants and transition relations, which are often presented in various logics supported by different specification languages.

To glimpse at the power of symbolic model checking, let us return to Figure 4.2. Recall that TLC's performance rapidly decreased, as N increased, due to the fact that it had to compute N^2 distinct, independent explicit states, each reachable from any other. In Table 4.2 we show how Apalache, which uses a symbolic approach, explores this same specification. We already know that any reachable state of the specification is reachable in a single step, so we need only execute a single symbolic transition with Apalache.

Notice that the Apalache runtime is constant, and in particular independent of N. This is possible, because membership in an integer interval, $x \in a..b$, can be represented as two constraints: $x \geq a \wedge x \leq b$. If we were to instead select from a non-uniform set (instead of $1..N$) as the transition step, the number of Apalache constraints would scale approximately linearly with the size of the set. The number of constraints describing the transitions is constant, the only bottleneck is the number of constraints required to encode the set from which elements are selected. See 7 for a detailed explanation of set encodings in Apalache.

CHAPTER 5

State of the art in verification

In this chapter, we give an overview of the current landscape of tools used in the formal verification of specifications in various languages.

5.1 TLC

The TLC model checker, introduced in [YML99] was one of the first tools designed specifically for TLA$^+$. The idea behind its design was simple: instead of tackling the analysis of a system implementation, which may be very large, and the nuances of which may be very complicated, many bugs can be found more quickly be analyzing a system specification. As mentioned before, system specifications, unlike implementations, are written at a much higher level. In particular, specifications written in TLA$^+$, rely heavily on expressions in, for example, first-order logic an Zermelo-Fränkel set theory. This proved to pose quite a challenge to the tool designers, which had to strike a balance between maintaining the expressiveness of the TLA$^+$ language fragment TLC would support, and the tool's ability to successfully compute models in reasonable time, or in fact at all.

TLC today, with over two decades of engineering effort, is part of a larger package of TLA$^+$-related software called the TLA$^+$ Toolbox [KLR19b], maintained by Microsoft Research. The underlying technique used in TLC, explicit state model checking is described in Section 4.1. Despite the relative simplicity of the approach, there have been numerous case studies over the years, describing the use of TLA$^+$, and TLC, in bug-finding efforts in some of the worlds leading technology companies. TLC has played a role in finding a bug in Compaq's multiprocessor [YML99], verifying cache-coherence protocols [JLM$^+$03], and is used by Amazon Web Services to check properties of their cloud-storage architecture [NRZ$^+$15].

5.2 TLAPS

Part of the same TLA$^+$ Toolbox, the TLA$^+$ proof system (TLAPS) [TLAb], introduced in [CDLM10] offers an alternative avenue for the verification of TLA$^+$ specifications. Unlike TLC, TLAPS facilitates the writing of proofs and formal reasoning within TLA$^+$, about the various formulas in a specification. Writing proofs, as opposed to model-checking has both benefits and drawbacks: On the one hand, for the purposes of a proof, it is often irrelevant whether an object (set, sequence, function, etc.) is finite or infinite, as long as it satisfies a certain property. This means that proofs can reason about, for example, the actual (infinite) set of integers, as opposed to a finite set of bounded integers. Moreover, proofs tend to suffer less from nondeterminism in specifications, as the state-space explosion often caused by nondeterminism in explicit-state model checking does not happen with proofs. On the other hand, proof-writing is much more labor intensive; while model checking an already written specification requires next to no input from the specification author, as tools such as TLC are non-interactive in their state-space exploration, writing a proof requires significant additional effort. Consequently, this creates a barrier to entry, as specification authors often have engineering backgrounds, but lack experience with formal-proof writing. An example of a TLAPS proof can be found in [CDL$^+$12], where it is used to prove that mutual exclusion is guaranteed in Peterson's algorithm. Notably, TLAPS also uses SMT internally, but in a different fashion, compared to model checkers, with an assortment of engines: Z3 [DB08], CVC4 [DRK$^+$14], as well as the non-SMT based Zenon [BDD07] and Isabelle [NPW02].

Ultimately, both techniques are best used in tandem; model-checking allows for relatively quick assurances and error detection, better facilitating the writing of a specification from the ground up, while proof-writing provides air-tight guarantees of system correctness, for more matured specifications of systems possibly too complex to have their states enumerated in reasonable time. To better understand how Apalache, TLAPS, and TLC work together, see [KKM22].

5.3 Alloy

TLA$^+$ is not the only formalism used in system specification. One of its counterparts is the specification language Alloy [Jac02]. Unlike TLA$^+$, the strength of which lies in expressing temporal behavior, i.e. the evolution of a system over time, Alloy's core design lends itself better to describing structural properties; the fundamental building blocks of Alloy specifications are relations, not temporal formulas.

A comparison between the two can be found in [MC16], but in short, TLA$^+$ is better suited for describing the behavior of a system, while Alloy is better suited for describing the static structure of a system, though Alloy has recently been extended with support for temporal properties.

In a parallel to TLA$^+$'s TLC, Alloy has its own associated model checker, the Alloy Analyzer [JSS00]. Interestingly, the analyzer internally uses SAT encodings of Alloy

Figure 5.1: Alloy specification of a filesystem

```
1  sig Name {}
2  abstract sig Obj {
3    name : one Name
4  }
5  fact {
6    all n : Name | lone name · n
7  }
8  sig Dir, File extends Obj{}
9  sig FS {
10   objects : set Obj,
11   root : one (Dir & objects),
12   parent : objects → lone (Dir & objects)
13 }
14 assert {
15   all fs : FS | some fs · objects
16 }
17 pred cd [f, f' : FS, d : Dir]{...}
18 pred mv [f, f' : FS, o : Object, d:Dir]{...}
```

formulas, which it offloads to third party SAT solvers. In this sense, the Alloy analyzer shares design similarities with the scope of this work, since our goal is to translate TLA$^+$ into SMT and offload the work to established SMT solvers (which also rely on SAT internally).

Figure 5.1 demonstrates an example Alloy specification of a filesystem, taken from [CP09].

The Alloy website [all21] provides a comprehensive list of case studies, referencing a plethora of industrial use-cases.

5.4 The B-method ecosystem

The B-method, introduced in [ALN$^+$91], is a formalism for the development of computer systems, centered around the notions of abstract machines and refinement. It uses predicate logic to specify machine invariants. Proofs in the formalism mostly include consistency checking, i.e. verifying that the provided invariant holds in all machine-reachable states, and refinement checking, i.e. verifying that one machine is refined by another, in which the state is represented by more concrete data structures, or the operations of which are more deterministic.

It has a variety of associated tools: Atelier-B [Lec14], B-toolikit [HL96, Rob97] and ProB [LB03], which generate predicate-logic proof obligations, towards proving consistency or

refinement. Both automatic- and interactive-proving is possible, as well as automatic translation of low-level B specifications into executable code.

Industrial use of B includes cases in the automotive industry [LB03], tourism [LB03], transportation [CDP+17], and aeronautics research [SA17].

In relation to TLA$^+$, there exists a translator, TLA2B [HBL14], which allows the translation of a subset of TLA$^+$ into B. Notably, as B is typed, the translation fails on TLA$^+$ specifications which include nonhomogeneous collections (e.g. sets) or contain variables, the types of which are not fixed. It also lacks support for temporal operators and recursive definitions, which makes its use limited in practice.

5.5 Z notation

Z-notation is a specification language, based on set theory, lambda calculus and first order predicate logic. Unlike most of the other languages mentioned in this section, Z-notation is a typed language, meaning that every Z-expression has an associated type. Fundamental building-blocks of Z specifications are called schemas; each schema defines either a description of permissible states or changes thereof, invariants, possible operations, or input-output relations. This focus makes Z well-suited for reasoning about implementations, since, for example, the operations in Z are a natural way of reasoning about function calls in executable languages, in terms of pre- and post-conditions.

A comparison between Z and B can be found in [KGS12]. Like B, Z-notation has been used in the railway industry [Zaf09], as well as in cloud-security [BSM16].

5.6 Type retrofitting

Similar to how we introduce a type system for TLA$^+$, we have recently seen several initiatives to retrofit existing untyped (programming) languages with type systems. The first one of note is TypeScript [BAT14, RSF+15], which, among other things, extends JavaScript with a static type system, as well as type inference. An example can be found in Figure 5.2.

Interestingly, their type system is unsound, meaning that some type errors may elude static analysis and result in run-time exceptions. The authors argue, in our opinion correctly, that type-soundness is not a prerequisite to usefulness, as many type-errors are still caught by this unsound analysis. Similar initiatives exist for other languages as well, for example Python [OPSR15, VKSB14].

Their approaches are undoubtedly useful to analyze, but unfortunately cannot be directly replicated for TLA$^+$. The reason for this is relatively simple: the languages they are extending are programming languages, designed to be executable without any kind of type annotations, and the type systems only ever serve to provide safety guarantees, or preempt run-time exceptions. This means that it is perfectly valid to type-annotate or

Figure 5.2: A TypeScript snippet, with an embedded error it would produce

```
interface Student {
  name: string;
  id: number;
}

const student: Student = {
  surname: "Smith",

/*  Type '{ surname: string; id: number; }' is not
    assignable to type 'Student'.
    Object literal may only specify known properties,
    and 'surname' does not exist in type 'Student'. */

  id: 0,
};
```

analyze only parts of the input, e.g. an especially critical function. In our case, TLA$^+$ is not an executable language by design. Because Apalache relies on an SMT encoding internally, there can be no partial typing — every single expression in the specification must be type-annotated (or typeable by means of automatic type inference). So while a type system is a quality-of-life improvement in the JavaScript case, it is a necessity in the TLA$^+$ case. Note that this is even implicitly true for TLC, which does not have a formal type system. For example, non-homogeneous collections, such as $\{1, "abc"\}$ are rejected by TLC (as well as Apalache).

5.7 ByMC

The Byzantine Model Checker (ByMC) [KW18, KLSW20] implements techniques for verifying threshold-guarded distributed algorithms, by means of threshold automata, presented in [KLVW17b, KVW15]. It was the first tool to automatically verify several fault-tolerant algorithms in the parametric setting [BGMR01, DS06, Gue02, MMPR03, Ray97, SvR08]. The tool works in the following way: As input, it requires a threshold automaton with bounded diameter [KVW14], or a specification in Parametric Promela, an extension of the Promela language used by the Spin model checker [Hol03], adapted to allow specifications of infinite-state (parametrized) systems. A discussion on the benefits and drawbacks of both input formats can be found in [KW18]. From either input the relevant safety/liveness properties are extracted in a temporal logic fragment ELTL_{FT}, described in [KW18]. Each possible counterexample, i.e. violation of one of the specification properties, can have one of finitely many shapes, and all such shapes are enumerated by the tool. For each shape, the tool constructs an SMT query, encoding

the existence of a counterexample of that shape, which is offloaded to an external SMT solver. If no counterexample is found, the property holds, otherwise a counterexample is reported (finite trace, in cases of safety violation, or lasso for liveness). An example of a threshold-automaton input can be found in Figure 5.3. The full automaton and its Parameterized Promela equivalent can be found in [ben].

5.8 Ivy

Ivy [PMP+16] refers to a language and a toolkit for protocol specification and modeling. Using Ivy, one can construct a discrete transition system, from data, functions and actions. On top of those, one can specify various invariants, from simple state properties, to safety and liveness. It also has built-in language features, called monitors, which are designed to facilitate refinement, by modifying actions with e.g. pre- and post-effects. It is a typed language, meaning that every data variable and parameter must have a declared type, though there is support for uninterpreted types.

A recent summary [MP20] outlines the success of the language in also generating performant executable code, directly from a specification.

While Ivy is similar to TLA$^+$ in many aspects, there are also notable differences, the first being the presence of types. In addition, the Ivy prover can reason about infinite-state systems (which TLC cannot, and Apalache can, in certain cases). As far as syntax is concerned, like many other languages in this section, it does not have complex data structures built into the language itself, though, unlike TLA$^+$, it allows for the definition of abstract datatypes, much like modern programming languages. Most notably, it lacks built-in support for sets, which sometimes makes it difficult to directly compare or translate between Ivy and TLA$^+$.

More information about Ivy, as well as a tutorial with examples, can be found on their Github page [ivy].

Figure 5.3: A threshold automaton for reliable broadcast [ST87] (excerpt)

```
skel Proc {
  local pc;
  shared nsnt;
  parameters N, T, F;

  define THRESH1 == T + 1;
  define THRESH2 == N - T;

  assumptions (0) {
    N > 3 * T;
    T >= F;
    T >= 1;
  }

  locations (0) {
    loc0: [0];
    loc1: [1];
    ...
  }

  inits (0) {
    (loc0 + loc1) == N - F;
    locSE == 0;
    locAC == 0;
    nsnt == 0;
  }

  rules (8) {
  0: loc1 -> locSE
      when (true)
      do { nsnt' == nsnt + 1; };
  1: loc0 -> locAC
      when (nsnt >= THRESH2 - F)
      do { nsnt' == nsnt + 1; };
  ...
  }

  specifications (0) {
    ...
    corr: <>[]((nsnt < THRESH1 || loc0 == 0)
                && (nsnt < THRESH2 || loc0 == 0)
                && ((nsnt < THRESH2) || locSE == 0)
                && (loc1 == 0))
        -> ((loc0 == 0) -> <>(locAC != 0));
    ...
  }
}
```

CHAPTER 6

Symbolic transitions in TLA$^+$

This chapter is an extension of the work presented in [KTK18]. The techniques described in it were implemented by the author in the Apalache model checker.

6.1 Introduction

A simple example in Figure 6.1 illustrates the problems that one faces when developing a symbolic model checker for TLA$^+$. In this example, we model two processes: *Producer* that inserts a subset of {"A", "B", "C", "D", "E"} into the set S, and *Consumer* that removes from S its arbitrary subset. The system is initialized with the operator *Init*. A system transition is specified with the operator *Next* that is defined via a disjunction of operators *Produce* and *Consume*. Both Producer and Consumer maintain the state invariant $empty \Leftrightarrow (S = \emptyset)$. We notice the following challenges for a symbolic approach:

1. The specification does not have types. This is not a problem for TLC, since it constructs states on the fly and hence dynamically computes types. In the symbolic case, one can use type synthesis [MV12a] or the untyped SMT encoding [MV12b].
2. Direct translation of *Next* to SMT would produce a *monolithic* formula, e.g., it would not analyze *Produce* and *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 updates.

In this chapter, we focus on the second problem, and return to the first in Chapter 8. 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 — which would be anyway impossible without types — and evaluating *Next* on them, TLC explores subformulas of *Next*. The essential exploration rules are: (1) Disjunctions and

MODULE *prodcons*

VARIABLE S, *empty*

$$\begin{aligned}
&\mathit{Init} \triangleq S = \{\} \land \mathit{empty} = \text{TRUE} \\
&\mathit{Produce} \triangleq \land \mathit{empty}' = \text{FALSE} \\
&\qquad\qquad \land \exists X \in \text{SUBSET } \{\text{“A”}, \text{“B”}, \text{“C”}, \text{“D”}, \text{“E”}\} : S' = S \cup \{X\} \\
&\mathit{Consume} \triangleq \neg \mathit{empty} \land S' \in \text{SUBSET } S \land \mathit{empty}' = (S' = \{\}) \\
&\mathit{Next} \triangleq \mathit{Produce} \lor \mathit{Consume}
\end{aligned}$$

Figure 6.1: A simple producer-consumer

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 *Produce* and *Consume* independently and assigns variables as prescribed by these formulas. As TLC is explicit, for each state, it produces at most 2^{2^5} successors in *Produce* as well as in *Consume*.

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

1. For purely Boolean formulas, if ϕ is transformed into an equivalent formula in disjunctive normal form (DNF): $\bigvee_{1 \le i \le k} D_i$, then every disjunct D_i has *exactly one* assignment per variable from V'.
2. The assignments adhere 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'.
3. 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) $\mathit{empty}' = \text{TRUE}$, (2) $S' = S \cup \{X\}$, (3) $S' \in \text{SUBSET } S$, and (4) $\mathit{empty}' = (S' = \emptyset)$. 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 experiments.

Our contributions are as follows:

- We formalize the notion of assignments and assignment strategies, which can be used in model checking, and present an SMT encoding from which assignment strategies are easily extracted.

- We define a sound decomposition of a TLA$^+$ formula into symbolic transitions using assignment strategies, which enables a modular analysis of the original formula.

- We implement the above as part of the APALACHE model checker [pro], and present experimental results using several state-of-the-art TLA$^+$ specifications, including complex algorithms such as Paxos or RAFT.

We have also conducted experiments on over 30 TLA$^+$ benchmarks, which we report on in Section 6.7.

The chapter is organized as follows: Section 6.2 introduces an abstraction of TLA$^+$ syntax, called α-TLA$^+$, which preserves only those language constructs, that are useful for determining assignments. In Section 6.3, we introduce auxiliary notions, such as label sets, assignment candidates and the dependency relation. Section 6.4 introduces branches – Boolean formulas abstracting the structure of α-TLA$^+$ – and the definition of an assignment strategy, in terms of its branch properties. Section 6.5 presents the encoding of assignment strategies into SMT. In Section 6.6, 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 6.7 details experimental results and Section 6.8 contains concluding remarks.

6.2 Abstract syntax α-TLA$^+$

TLA$^+$ has rich syntax [Lam02], which cannot be defined here. 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 $\star$.

We assume predefined three infinite sets:

- A set $\mathcal{L}$ of *labels*. We use notation ℓ_i to refer to its elements, for $i \in \mathbb{N}$.

- A set $Vars'$ of *primed variables* that are decorated with prime, e.g., x' and a'.

- A set *Bound* of *bound variables*, which are used by quantifiers.

$$Next \triangleq \ell_1 :: \Big(\ell_2 :: (\ell_3 :: empty' \in \ell_4 :: \star \wedge \ell_5 :: \exists\, X \in \ell_6 :: \star : \ell_7 :: S' \in \ell_8 :: \star)$$
$$\vee\, \ell_9 :: (\ell_{10} :: \star \wedge \ell_{11} :: S' \in \ell_{12} :: \star \wedge \ell_{13} :: empty' \in \ell_{14} :: \star(S'))\Big)$$

Figure 6.2: The *Next* operator of producer-consumer in α-TLA$^+$

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

$$\begin{aligned}
expr &::= ex_\alpha \mid \ell :: \text{FALSE} \\
&\quad \mid \ell :: v' \in ex_\alpha \mid \ell :: expr \wedge \cdots \wedge expr \mid \ell :: expr \vee \cdots \vee expr \\
&\quad \mid \ell :: \exists x \in ex_\alpha :\ expr \mid \ell :: \text{IF } ex_\alpha \text{ THEN } expr \text{ ELSE } expr \\
ex_\alpha &::= \ell :: \star(v', \ldots, v') \\
\ell &::= \text{a unique label from the set } \mathcal{L} \\
v' &::= \text{a variable name from the set } Vars' \\
x &::= \text{a variable name from the set } Bound
\end{aligned}$$

A few comments on the syntax and its relation to TLA$^+$ expressions are in order. We require every expression to carry a unique label $\ell_i \in \mathcal{L}$. 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 $\langle v_1, \ldots, v_k \rangle$ 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 $\langle v_1, \ldots, v_k \rangle$ as $v_1' \in \{v_1\} \wedge \cdots \wedge v_k' \in \{v_k\}$. Every non-essential TLA$^+$ expression e is presented in the abstract form $\ell :: \star(v_1', \ldots, v_k')$, 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 $\ell :: \star$. 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 $\star$; 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 α-TLA$^+$, that represents an expression e in the complete TLA$^+$ syntax. With γ we denote the reverse translation from α-TLA$^+$ to TLA$^+$ that has the property that $\gamma(\alpha(e)) = e$. Figure 6.2 shows the abstract expression $\alpha(Next)$ of the operator *Next* defined in Figure 6.1.

Discussions Notice that α-TLA$^+$ is missing several fundamental constructs permitted in TLA$^+$, such as CASE expressions, universal quantifiers, and negations. They are all abstracted to $\star$. The primary purpose of α-TLA$^+$ is to allow us to determine whether a

Table 6.1: The definition of $\mathrm{Sub}(\phi)$

α-**TLA$^+$ expression** ϕ	$\mathrm{Sub}(\phi)$
$\ell :: \text{FALSE}$ or $\ell :: \star(v'_1, \ldots, v'_k)$	$\{\phi\}$
$\ell :: v' \in \phi_1$	$\{\phi, \phi_1\}$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$ or $\ell :: \bigvee_{i=1}^{s} \phi_i$	$\{\phi\} \cup \bigcup_{i=1}^{s} \mathrm{Sub}(\phi_i)$
$\ell :: \exists x \in \phi_1 : \ \phi_2$	$\{\phi\} \cup \mathrm{Sub}(\phi_1) \cup \mathrm{Sub}(\phi_2)$
$\ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$	$\{\phi\} \cup \mathrm{Sub}(\phi_1) \cup \mathrm{Sub}(\phi_2) \cup \mathrm{Sub}(\phi_3)$

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 similar reason, we abstract the expressions under negation. The latter is consistent with TLC, which produces an error when given, for example, $\mathit{Next} == \neg(x' = 1)$. 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.

6.3 Preliminary definitions

Every TLA$^+$ specification declares a certain finite set of variables, which may appear in the formulas contained therein. Let ϕ be an α-TLA$^+$ expression. We assume, for the purposes of our analysis, that ϕ is associated with some finite set $\mathit{Vars}'(\phi)$, which is a subset of Vars', containing all of the variables that appear in ϕ (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 $\mathrm{lab}(\ell :: \psi)$ to refer to the expression label ℓ and $\mathrm{expr}(\ell)$ to refer to the expression that is labeled with ℓ. We refer to the set of all subexpressions of ϕ by $\mathrm{Sub}(\phi)$. See Table 6.1 for a formal definition.

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

Of special interest to us are *assignment candidates*, i.e., expressions of the form $\ell :: v' \in \phi_1$. Given an arbitrary variable $v' \in \mathit{Vars}'(\phi)$ and an α-TLA$^+$ expression ϕ, we write $\mathrm{cand}(v', \phi)$ to mean the set of labels that belong to assignment candidates for v' in subexpressions of ϕ. More formally, $\mathrm{cand}(v', \phi)$ is $\{\ell \mid (\ell :: v' \in \psi) \in \mathrm{Sub}(\phi)\}$. An exhaustive definition can be found in Table 6.2. We use the notation $\mathrm{cand}(\phi)$ to mean $\bigcup_{v' \in \mathit{Vars}'(\phi)} \mathrm{cand}(v', \phi)$.

Finally, we assign to each label ℓ in $\mathrm{Labs}(\phi)$ a set $\mathrm{frozen}_\phi(\ell) \subseteq \mathit{Vars}'(\phi)$. Intuitively, if a variable v' is in $\mathrm{frozen}_\phi(\ell)$, then no expression of the form $\hat{\ell} :: v' \in \psi$ can be treated as an assignment inside $\mathrm{expr}(\ell)$. Formally, for every $\ell \in \mathrm{Labs}(\phi)$ the set $\mathrm{frozen}_\phi(\ell)$ is defined as the minimal set satisfying all the constraints in Table 6.3.

Table 6.2: The definition of $\text{cand}(v', \phi)$

α-**TLA$^+$ expression** ϕ	$\text{cand}(v', \phi)$
$\ell :: \text{FALSE}$ or $\ell :: \star(v'_1, \ldots, v'_k)$	$\emptyset$
$\ell :: w' \in \phi_1$	$\begin{cases} \{\ell\} & ; w' = v' \\ \emptyset & ; \text{otherwise} \end{cases}$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$ or $\ell :: \bigvee_{i=1}^{s} \phi_i$	$\bigcup_{i=1}^{s} \text{cand}(v', \phi_i)$
$\ell :: \exists x \in \phi_1 : \ \phi_2$	$\text{cand}(v', \phi_2)$
$\ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$	$\text{cand}(v', \phi_2) \cup \text{cand}(v', \phi_3)$

Table 6.3: The constraints on frozen_ϕ

α-**TLA$^+$** ϕ	**Constraints on** frozen_ϕ
$\ell :: \star(v'_1, \ldots, v'_k)$	$\{v'_1, \ldots, v'_k\} \subseteq \text{frozen}_\phi(\ell)$
$\ell :: v' \in \phi_1$	$\text{frozen}_\phi(\ell) = \text{frozen}_\phi(\text{lab}(\phi_1))$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$ $\ell :: \bigvee_{i=1}^{s} \phi_i$	$\text{frozen}_\phi(\ell) \subseteq \text{frozen}_\phi(\text{lab}(\phi_i))$ for $i \in \{1, \ldots, s\}$
$\ell :: \exists x \in \phi_1 : \ \phi_2$	$\text{frozen}_\phi(\ell) \subseteq \text{frozen}_\phi(\text{lab}(\phi_1)) \subseteq \text{frozen}_\phi(\text{lab}(\phi_2))$
$\ell :: \text{IF } \phi_1 \text{ THEN } \phi_2$	$\text{frozen}_\phi(\ell) \subseteq \text{frozen}_\phi(\text{lab}(\phi_1))$
$\text{ELSE } \phi_3$	$\text{frozen}_\phi(\text{lab}(\phi_1)) \subseteq \text{frozen}_\phi(\text{lab}(\phi_i))$ for $i = 2, 3$

The sets frozen_ϕ naturally lead to the dependency relations $\lhd_{v'}$ on $\text{Labs}(\phi)$, where $v' \in \mathit{Vars}'(\phi)$. We will use $\ell_1 \lhd_{v'} \ell_2$ to mean that ℓ_1 is an assignment candidate for v', which also belongs to the frozen set of ℓ_2. Formally:

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

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

Example 1. *Let us look at the following α-TLA$^+$ expression:*

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

Take the subexpression $\ell_3 :: x' \in [\ell_4 :: \star]$, which we name ψ . By solving the constraints for $\text{frozen}_\psi(\ell_3)$ we conclude that $\text{frozen}_\psi(\ell_3) = \emptyset$. 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 ℓ_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 ψ, implicitly depend on the bound variable and the expressions used to define it, which is $\text{expr}(\ell_2)$ in our example.

Table 6.4: The definition of boolForm(ϕ)

α-**TLA$^+$ expression** ϕ	boolForm(ϕ)
ℓ :: FALSE or ℓ :: $\star(v'_1, \ldots, v'_k)$ or ℓ :: $v' \in \phi_1$	b_ℓ
ℓ :: $\bigwedge_{i=1}^{s} \phi_i$	$\bigwedge_{i=1}^{s}$ boolForm(ϕ_i)
ℓ :: $\bigvee_{i=1}^{s} \phi_i$	$\bigvee_{i=1}^{s}$ boolForm(ϕ_i)
ℓ :: $\exists x \in \phi_1 : \ \phi_2$	boolForm(ϕ_2)
ℓ :: IF ϕ_1 THEN ϕ_2 ELSE ϕ_3	boolForm(ϕ_2) $\vee$ boolForm(ϕ_3)

6.4 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 – boolForm (ϕ) is a Boolean abstraction of an α-TLA$^+$ formula ϕ. 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 α-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 6.4.

As boolForm(ϕ) is a formula in Boolean logic, a model of boolForm(ϕ) 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 $\mathcal{M}[S] \vDash b_\ell$ 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 ϕ if the following constraints hold:

(a) The set Br induces a model of boolForm(ϕ), i.e., $\mathcal{M}[Br] \vDash$ boolForm(ϕ), and

(b) The model $\mathcal{M}[Br]$ is minimal, that is, $\mathcal{M}[S] \nvDash$ boolForm(ϕ) for every $S \subset Br$.

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

Example 2. *Let us look the α-TLA$^+$ expression ϕ given by*

$$\ell_1 :: [[\ell_2 :: x' \in \star] \wedge [\ell_3 :: [[\ell_4 :: x' \in \star] \vee [\ell_5 :: x' \in \star]]]]$$

We know that $\text{boolForm}(\phi) = b_{\ell_2} \wedge (b_{\ell_4} \vee b_{\ell_5})$. *The set* $S = \{\ell_2, \ell_4, \ell_5\}$ *induces a model of* $\text{boolForm}(\phi)$, *but it is not a branch of* ϕ *because* $\mathcal{M}[S]$ *is not a minimal model. It is easy to see that* ϕ *has two branches* $Br_1 = \{\ell_2, \ell_4\}$, *and* $Br_2 = \{\ell_2, \ell_5\}$. *Therefore, we see that* $\text{Branches}(\phi) = \{Br_1, Br_2\}$.

6.4.1 Properties of induced models

To better understand boolForm, we need to look at how, for a fixed formula ϕ, the choice of S influences whether $\mathcal{M}[S] \vDash \text{boolForm}(\phi)$ or not.

The first property, that might appear obvious, but is in fact quite useful, is that labels outside of $\text{Labs}(\phi)$ have no bearing on whether or not an induced model satisfies $\text{boolForm}(\phi)$. Formally, the following holds true:

Lemma 1. *Let* ϕ *be an* α*-TLA*$^+$ *expression. For any set* $A \subseteq \mathcal{L}$, *it holds that*

$$\mathcal{M}[A] \vDash \text{boolForm}(\phi) \iff \mathcal{M}[A \cap \text{Labs}(\phi)] \vDash \text{boolForm}(\phi)$$

Proof. We prove this by induction on the structure of ϕ:

- $\phi = \ell :: \text{FALSE}$: By definition, $\text{boolForm}(\phi) = b_\ell$ and $\text{Labs}(\phi) = \{\ell\}$. It is clear that $\ell \in A \iff \ell \in A \cap \{\ell\}$. If we look at the definition of the induced model, we can conclude the following:
$$\mathcal{M}[A] \vDash b_\ell \iff \ell \in A \iff \ell \in A \cap \{\ell\} \iff \mathcal{M}[A \cap \{\ell\}] \vDash b_\ell$$
- $\phi = \ell :: \star(v'_1, \ldots, v'_k)$: Same as for $\phi = \ell :: \text{FALSE}$.
- $\phi = \ell :: v' \in \hat{\ell} :: \star(v'_1, \ldots, v'_k)$: By definition, $\text{Labs}(\phi) = \{\ell, \hat{\ell}\}$. Again, $\ell \in A \iff \ell \in A \cap \text{Labs}(\phi)$, the rest is the same as for $\phi = \ell :: \text{FALSE}$.
- $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$: Assume as the induction hypothesis, that the lemma holds for every $\phi_i, i \in \{1, \ldots, s\}$. As $\text{boolForm}(\phi) = \bigwedge_{i=1}^{s} \text{boolForm}(\phi_i)$ by definition, we know that
$$\mathcal{M}[A] \vDash \text{boolForm}(\phi) \iff \mathcal{M}[A] \vDash \text{boolForm}(\phi_i), \text{ for all } i \in \{1, \ldots, s\}$$
Take an arbitrary $i \in \{1, \ldots, s\}$. By the induction hypothesis
$$\mathcal{M}[A] \vDash \text{boolForm}(\phi_i) \iff \mathcal{M}[A \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi_i)$$
By applying the hypothesis again, it is also the case that
$$\begin{aligned} &\mathcal{M}[A \cap \text{Labs}(\phi)] \vDash \text{boolForm}(\phi_i) \\ \iff &\mathcal{M}[(A \cap \text{Labs}(\phi)) \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi_i) \end{aligned}$$
Since $\text{Labs}(\phi) \cap \text{Labs}(\phi_i) = \text{Labs}(\phi_i)$ we can conclude that
$$\mathcal{M}[A] \vDash \text{boolForm}(\phi_i) \iff \mathcal{M}[A \cap \text{Labs}(\phi)] \vDash \text{boolForm}(\phi_i)$$
Since i is arbitrary, this holds for every ϕ_i, so the lemma holds for such ϕ.

- $\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$: Analogous to the case where $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$.
- $\phi = \ell :: \exists x \in \psi : \phi_0$: Assume as the induction hypothesis, that the lemma holds for ϕ_0. Since $\text{Labs}(\phi_0) \subseteq \text{Labs}(\phi)$ it follows that $\text{Labs}(\phi_0) = \text{Labs}(\phi) \cap \text{Labs}(\phi)$. As $\text{boolForm}(\phi) = \text{boolForm}(\phi_0)$ by definition, we know

$$\begin{aligned}
\mathcal{M}[A] \vDash \text{boolForm}(\phi) &\iff \mathcal{M}[A] \vDash \text{boolForm}(\phi_0) \\
&\iff \mathcal{M}[A \cap \text{Labs}(\phi_0)] \vDash \text{boolForm}(\phi_0) \\
&\iff \mathcal{M}[A \cap \text{Labs}(\phi) \cap \text{Labs}(\phi_0)] \vDash \text{boolForm}(\phi_0) \\
&\iff \mathcal{M}[A \cap \text{Labs}(\phi)] \vDash \text{boolForm}(\phi_0) \\
&\iff \mathcal{M}[A \cap \text{Labs}(\phi)] \vDash \text{boolForm}(\phi)
\end{aligned}$$

- $\phi = \ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$: Analogous to the case where $\phi = \ell :: \phi_2 \vee \phi_3$ as $\text{boolForm}(\phi) = \text{boolForm}(\phi_2) \vee \text{boolForm}(\phi_3)$.

Thus the lemma holds for any α-TLA$^+$ expression ϕ. □

The next lemma highlights what could be considered the monotonicity of induced models. For a fixed formula ϕ, its structural formula $\text{boolForm}(\phi)$ is a purely propositional formula, so we can always consider it to be in negation-normal form (NNF). It turns out, that boolForm has only positive atoms and no negation, and is thus trivially in NNF already. Consequently, the NNF form of $\neg\,\text{boolForm}(\phi)$ has all atoms negated. Therefore, if a set of labels S induces a model that satisfies $\neg\,\text{boolForm}(\phi)$, so must any subset of S, because removing a label ℓ from S means that the new set induces a model of a greater number of negative atoms (particularly, $\neg b_\ell$). Formally:

Lemma 2. *For a propositional formula ψ in NNF, which contains only negated atoms $\neg b_{\ell_1}, \ldots, \neg b_{\ell_k}$, and $S \subseteq \mathcal{L}$ such that $\mathcal{M}[S] \vDash \psi$ it holds that $\mathcal{M}[J] \vDash \psi$, for every $J \subseteq S$.*

Proof. We can prove this by induction on the structure of ψ:

- $\psi = \neg b_\ell$: By definition,
$$\mathcal{M}[S] \vDash \neg b_\ell \iff \ell \notin S$$
Since $J \subseteq S$, we know $\ell \notin S$ implies $\ell \notin J$. Thus, $\mathcal{M}[J] \vDash \neg b_\ell$.
- $\psi = \bigwedge_{i=1}^{s} \psi_i$: Assume as the induction hypothesis, that the lemma holds for all ψ_i. We know
$$\mathcal{M}[S] \vDash \psi \iff \mathcal{M}[S] \vDash \psi_i, \text{ for all } i \in \{1, \ldots, s\}$$
If $\mathcal{M}[S] \vDash \psi$ and $J \subseteq S$ it follows from the induction hypothesis, that $\mathcal{M}[J] \vDash \psi_i$ for all i. So clearly, $\mathcal{M}[J] \vDash \psi$.
- $\psi = \bigvee_{i=1}^{s} \psi_i$: Analogous to the previous case.

We conclude that the lemma holds for any propositional formula ψ in NNF. □

6.4.2 Computing branches recursively

Previously, we gave the formal definition of a branch (and set of branches), in terms of the minimal models they induce for boolForm(ϕ). While this definition captures the properties we want from branches, it is not immediately obvious how these branches are found. To elaborate on the branch computation process, we introduce a series of lemmas, that demonstrate how to compute branches of an expression from the branches of its subexpressions.

In the case of conjunction, we obtain a branch of $\phi \wedge \psi$ by taking the union of any two branches $Br_\phi \cup Br_\psi$ where Br_ϕ is a branch of ϕ and Br_ψ is a branch of ψ. More generally:

Lemma 3. *Let ϕ be an α-TLA$^+$ expression. If ϕ has the shape $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$ it follows that every branch of ϕ is a union of branches for each ϕ_i and vice-versa. Formally:*

$$\text{Branches}(\phi) = \left\{ \bigcup_{i=1}^{s} Br_i \mid \forall i \in \{1, \ldots, s\} \;.\; Br_i \in \text{Branches}(\phi_i) \right\}$$

Proof. Take an arbitrary $Br \in \text{Branches}(\phi)$. By the definition of a branch, $\mathcal{M}[Br] \vDash \text{boolForm}(\phi)$. We define $Br_i \coloneqq Br \cap \text{Labs}(\phi_i)$ for each $i = 1, \ldots, s$. Then, $Br = \bigcup_{i=1}^{s} Br_i$ by construction, since $\text{Labs}(\phi) = \bigcup_{i=1}^{s} \text{Labs}(\phi_i)$. Because each subexpression of ϕ has a unique label, the sets $\text{Labs}(\phi_i)$ are pairwise disjoint.

It remains to be shown that each such Br_i is a branch, i.e. (a) each Br_i induces a model of boolForm (ϕ_i) and (b) no proper subset of Br_i induces such a model. Take an arbitrary $i \in \{1, \ldots, s\}$. Since boolForm(ϕ) implies boolForm(ϕ_i), we know $\mathcal{M}[Br] \vDash \text{boolForm}(\phi_i)$. By Lemma 1, it must be the case that $\mathcal{M}[Br_i] \vDash \text{boolForm}(\phi_i)$ as well, so point (a) holds.

To show (b) holds, take an arbitrary nonempty $T \subseteq Br_i$. Because Br induces a minimal model, we know $\mathcal{M}[Br \setminus T] \nvDash \text{boolForm}(\phi)$. If we look at any index $j \neq i$, since $\text{Labs}(\phi_i)$ and $\text{Labs}(\phi_j)$ are disjoint, the set $(Br \setminus T) \cap \text{Labs}(\phi_j)$ is just $Br \cap \text{Labs}(\phi_j) = Br_j$. Consequently, $\mathcal{M}[Br \setminus T] \vDash \text{boolForm}(\phi_j)$, by Lemma 1, for all $j \neq i$. However, $\mathcal{M}[Br \setminus T]$ does not model boolForm(ϕ). This must be because $\mathcal{M}[Br \setminus T] \nvDash \text{boolForm}(\phi_i)$. We can apply Lemma 1 again, to deduce that $(Br \setminus T) \cap \text{Labs}(\phi_i)$ also doesn't induce a model of $\text{boolForm}(\phi_i)$. Since $(Br \setminus T) \cap \text{Labs}(\phi_i)$ is $Br_i \setminus T$ and T is arbitrary, we see that for any $S \subset Br_i$ it must be the case that $\mathcal{M}[S] \nvDash \text{boolForm}(\phi_i)$, which proves that Br_i is indeed a branch of ϕ_i, for every $i \in \{1, \ldots, s\}$.

Alternatively, take arbitrary branches $Br_1, \ldots, Br_s$ of subexpressions, for which the following holds: $Br_1 \in \text{Branches}(\phi_1), \ldots, Br_s \in \text{Branches}(\phi_s)$. Define $Br \coloneqq \bigcup_{i=1}^{s} Br_i$. We must show that this Br is a branch of ϕ. Take an arbitrary $i \in \{1, \ldots, s\}$. By definition, $\mathcal{M}[Br_i] \vDash \text{boolForm}(\phi_i)$. Lemma 1 tells us that $\mathcal{M}[Br] \vDash \text{boolForm}(\phi_i)$ exactly when $\mathcal{M}[Br \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi_i)$. Because Br_i is minimal, it must be the case that $Br_i \cap \text{Labs}(\phi_i)$ equals Br_i. If it were some proper subset, $S \subset Br_i$, applying

Lemma 1 to Br_i would give us

$$\mathcal{M}[Br_i] \vDash \text{boolForm}(\phi_i) \iff \mathcal{M}[S] \vDash \text{boolForm}(\phi_i)$$

which contradicts the property that we know $\mathcal{M}[T] \nvDash \text{boolForm}(\phi_i)$ for every $T \subset Br_i$. It remains to be seen that $Br \cap \text{Labs}(\phi_i) = Br_i$. Expanding Br tells us

$$Br \cap \text{Labs}(\phi_i) = \bigcup_{j=1}^{s} Br_j \cap \text{Labs}(\phi_i)$$

If $i \neq j$ then, as $Br_j \subseteq \text{Labs}(\phi_j)$ and the label sets $\text{Labs}(\phi_i)$ and $\text{Labs}(\phi_j)$ are disjoint, we conclude $\text{Labs}(\phi_i) \cap Br_j = \emptyset$. So $\mathcal{M}[Br] \vDash \text{boolForm}(\phi_i)$. As i was arbitrary, this means $\mathcal{M}[Br] \vDash \bigwedge_{i=1}^{s} \text{boolForm}(\phi_i)$. To see that Br induces a minimal model, take an arbitrary nonempty $T \subseteq Br$. Then, $S \coloneqq Br \setminus T$ is a proper subset of Br. There must exist an i, for which $T \cap Br_i \neq \emptyset$. By Lemma 1, we know that

$$\begin{aligned}
\mathcal{M}[S] \vDash \text{boolForm}(\phi_i) &\iff \mathcal{M}[S \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi_i) \\
&\iff \mathcal{M}[(Br \setminus T) \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi_i) \\
&\iff \mathcal{M}[Br_i \setminus T] \vDash \text{boolForm}(\phi_i)
\end{aligned}$$

But Br_i is a branch and $Br_i \setminus T$ is its proper subset, so $\mathcal{M}[Br_i \setminus T] \nvDash \text{boolForm}(\phi_i)$ and consequently, $\mathcal{M}[S] \nvDash \text{boolForm}(\phi)$, for any proper subset $S \subset Br$. Therefore, Br is a branch of ϕ. □

In the case of disjunction, branch construction is even simpler. Suppose we take a branch Br of ϕ. Then, this same Br is also a branch of $\phi \vee \psi$, for any ψ. More generally:

Lemma 4. *Let ϕ be an α-TLA$^+$ expression. If ϕ has the shape $\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$ it follows that every branch of ϕ is a branch of some ϕ_i and vice-versa. Formally:*

$$\text{Branches}(\phi) = \bigcup_{i=1}^{s} \text{Branches}(\phi_i)$$

Proof. Take an arbitrary $i \in \{1, \ldots, s\}$ and $Br \in \text{Branches}(\phi_i)$. Since it is the case that $\mathcal{M}[Br] \vDash \text{boolForm}(\phi_i)$, it follows that $\mathcal{M}[Br] \vDash \bigvee_{j=1}^{s} \text{boolForm}(\phi_j)$. To see that Br is minimal, take an arbitrary $S \subset Br$. By definition, $\mathcal{M}[S] \nvDash \text{boolForm}(\phi_i)$. To see that it cannot induce a model for $\text{boolForm}(\phi_j)$, where $i \neq j$, we note that $\text{Labs}(\phi_i) \cap \text{Labs}(\phi_j) = \emptyset$ and, by Lemma 1,

$$\mathcal{M}[S] \vDash \text{boolForm}(\phi_j) \iff \mathcal{M}[S \cap \text{Labs}(\phi_j)] \vDash \text{boolForm}(\phi_j)$$

Since $S \subset \text{Labs}(\phi_i)$ we know that $S \cap \text{Labs}(\phi_j) = \emptyset$. As no boolForm formula is a tautology, by construction, it follows that $\mathcal{M}[\emptyset]$ cannot model $\text{boolForm}(\phi_j)$ for $j \neq i$. So S cannot induce a model for $\bigvee_{j=1}^{s} \text{boolForm}(\phi_j)$ and thus Br is a branch of ϕ.

Alternatively, take a $Br \in \text{Branches}(\phi)$. There must exist some $i \in \{1, \ldots, s\}$ for which $\mathcal{M}[Br] \vDash \text{boolForm}(\phi_i)$. We show that $Br \cap \text{Labs}(\phi_j) = \emptyset$ for all $i \neq j$ by contradiction: Assume that for some $j \neq i$ there exists a $x \in \text{Labs}(\phi_j) \cap Br$. It is always the case that $\text{Labs}(\phi_i)$ and $\text{Labs}(\phi_j)$ are disjoint. If we invoke Lemma 1, we see that

$$\mathcal{M}[Br] \vDash \text{boolForm}(\phi_i) \iff \mathcal{M}[Br \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi_i)$$

It must be the case that $Br \cap \text{Labs}(\phi_i)$ also induces a model for $\text{boolForm}(\phi_i)$ and therefore $\mathcal{M}[Br \cap \text{Labs}(\phi_i)] \vDash \text{boolForm}(\phi)$. But this is a contradiction, since Br is a branch and $Br \cap \text{Labs}(\phi_i)$ is a proper subset, since it doesn't contain x, which belongs to $\text{Labs}(\phi_j)$. Consequently, the assumption is false and $Br \cap \text{Labs}(\phi_j) = \emptyset$ for all $i \neq j$. It remains to see that no $S \subset Br$ can induce a model for $\text{boolForm}(\phi_i)$. Take an arbitrary $S \subset Br$. Since, for $i \neq j$, $Br \cap \text{Labs}(\phi_j) = \emptyset$ then $\mathcal{M}[Br] \nvDash \text{boolForm}(\phi_j)$. Because $\mathcal{M}[S] \nvDash \text{boolForm}(\phi)$, as Br is a branch, we must conclude that $\mathcal{M}[S] \nvDash \text{boolForm}(\phi_i)$. But that means Br is a branch of ϕ_i. □

As boolForm is defined in a way that it ignores quantification, it is not at all surprising that branches of quantified expressions are exactly branches of their respective bodies:

Lemma 5. *Let ϕ be an α-TLA$^+$ expression. If ϕ has the shape $\phi = \ell :: \exists x \in \psi.\ \phi_0$ it follows that branches of ϕ are exactly branches of ϕ_0. Formally:*

$$\text{Branches}(\phi) = \text{Branches}(\phi_0)$$

Proof. Clearly, as $\text{boolForm}(\phi) = \text{boolForm}(\phi_0)$ by definition, we know

$$\mathcal{M}[T] \vDash \text{boolForm}(\phi) \iff \mathcal{M}[T] \vDash \text{boolForm}(\phi_0)$$

for any $T \subseteq \mathcal{L}$, in particular also for branches. □

The last case is if-then-else splitting. Just as boolForm is translated as a disjunction of the boolForms for the then- and else- cases, so too is branch computation analogous to the disjunctive case.

Lemma 6. *Let ϕ be an α-TLA$^+$ expression. If $\phi = \ell$ :: IF ϕ_1 THEN ϕ_2 ELSE ϕ_3 it follows that every branch of ϕ is a branch of either ϕ_2 or ϕ_3 and vice-versa. Formally:*

$$\text{Branches}(\phi) = \text{Branches}(\phi_2) \cup \text{Branches}(\phi_3)$$

Proof. See the proof of Lemma 4, as $\text{boolForm}(\phi) = \text{boolForm}(\phi_2) \vee \text{boolForm}(\phi_3)$. □

6.4.3 Assignment strategies

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.

We want to statically mark some expressions as assignments, that is, pick a set $A \subseteq \text{Labs}(\phi)$. 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 1. *A set $H \subseteq \text{Labs}(\phi)$ is* homogeneous *if all the labels in H are assignment candidates. Formally, $H \subseteq \text{cand}(\phi)$.*

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* $d \in \mathbb{N}_0$, if some branch contains in H exactly d assignment candidates for the same variable. Formally, d is a covering index of $H \subseteq \text{Labs}(\phi)$, if we can find a branch $Br \in \text{Branches}(\phi)$ and a variable $v' \in \mathit{Vars}'(\phi)$ for which $d = |Br \cap H \cap \text{cand}(v', \phi)|$. A covering index of 0 means that some branch lacks assignments for at least one variable, and a covering index greater than 1 means that some branch contains more than one assignment to some variable. Now we define sets, that cover all branches with assignments:

Definition 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 $x' = y' \wedge y' = 2x'$. Its corresponding α-TLA$^+$ expression $\ell_0 :: (\ell_1 :: x' \in \ell_2 :: \star(y') \wedge \ell_3 :: y' \in \ell_4 :: \star(x'))$ has a minimal covering $\{\ell_1, \ell_3\}$. However, there is no way to order the assignments to x' and y'. To detect such cases, we define acyclic sets:

Definition 3. *A homogeneous set A is* acyclic *w.r.t. ϕ, if there exists a strict total order $\prec_A$ on A, with the following property: For every variable $v' \in V$, every branch $Br \in \text{Branches}(\phi)$ and every pair of labels ℓ_i and ℓ_j in $A \cap Br$ the relation $\ell_i \lhd_{v'} \ell_j$ implies $\ell_i \prec_A \ell_j$.*

In the previous example, $\{\ell_1, \ell_3\}$ is a minimal covering, but it is not acyclic; $\ell_1 \lhd_{x'} \ell_3$ and $\ell_3 \lhd_{y'} \ell_1$ produce the requirements $\ell_1 \prec_A \ell_3$ and $\ell_3 \prec_A \ell_1$, which are mutually exclusive for a strict total order.

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

Definition 4. *Let ϕ be an α-TLA$^+$ expression. A set $A \subseteq \mathcal{L}$ is an assignment strategy for ϕ, if it is an acyclic minimal covering.*

Now we are in a position to formulate the first problem we solve in this chapter:

Static assignment problem *Given an α-TLA$^+$ expression ϕ, our goal is to find an assignment strategy, or prove that none exists.*

6.5 Finding assignment strategies with SMT

For a given α-TLA$^+$ expression ϕ, we construct an SMT formula $\theta(\phi)$, that encodes the properties of assignment strategies. Technically, $\theta(\phi)$ is defined as $\theta_H(\phi) \wedge \theta_C(\phi) \wedge \theta_A(\phi)$, 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*).

Reasons to use SMT It turns out that computing syntactic properties, like whether or not an expression is an assignment candidate (for determining homogeneity and coverings) can be done easily in many different ways. The true value of SMT comes from being able to find acyclic sets, by solving LIA constraints. In this sense, SMT is a framework that allows us to easily encode all three conditions in the same language and extract assignment strategies directly from the produced solutions.

In the following, Propositions 1, 3, and 5 formally establish the relation between ϕ and its three SMT counterparts. Together, the propositions allow us to prove the following theorem:

Theorem 1. *For every α-TLA$^+$ formula ϕ and $A \subseteq \mathrm{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta(\phi)$ if and only if A is an assignment strategy for ϕ.*

6.5.1 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 $\mathcal{N}(\phi) \coloneqq \mathrm{Labs}(\phi) \setminus \mathrm{cand}(\phi)$. Then, we define the following:

$$\theta_H(\phi) \coloneqq \bigwedge_{\ell \,\in\, \mathcal{N}(\phi)} \neg b_\ell$$

Proposition 1. *For every α-TLA$^+$ expression ϕ and $A \subseteq \mathrm{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta_H(\phi)$ if and only if A is homogeneous.*

Proof. Firstly, assume $\mathcal{M}[A] \vDash \theta_H(\phi)$. Take an arbitrary $\ell \in \mathcal{N}(\phi)$. Then

$$\mathcal{M}[A] \vDash \theta_H(\phi) \Rightarrow \mathcal{M}[A] \vDash \neg b_\ell \iff \ell \notin A$$

So every element of A is in $\text{Labs}(\phi) \setminus \mathcal{N}(\phi) = \text{cand}(\phi)$, which means $A \subseteq \text{cand}(\phi)$, i.e. A is homogeneous.

Secondly, assume some $A \subseteq \text{Labs}(\phi)$ is homogeneous. Let ℓ be an arbitrary label in $\text{Labs}(\phi)$. The following must then be true:

$$\ell \in \mathcal{N}(\phi) \Rightarrow \ell \notin A \iff \mathcal{M}[A] \not\models b_\ell \iff \mathcal{M}[A] \models \neg b_\ell$$

So we can conclude that $\mathcal{M}[A] \models \bigwedge_{\ell \in \mathcal{N}(\phi)} \neg b_\ell$, that is, $\mathcal{M}[A] \models \theta_H(\phi)$. □

6.5.2 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 $v' \in \mathit{Vars}'(\phi)$, the transformation $\delta_{v'}$ as shown in Table 6.5. Intuitively, $\delta_{v'}(\phi)$ is satisfiable exactly when there is an assignment

Table 6.5: The definition of $\delta_{v'}(\phi)$

α-**TLA$^+$ expression** ϕ	$\delta_{v'}(\phi)$
ℓ :: FALSE or $\ell :: \star(v'_1, \ldots, v'_k)$	false
$\ell :: w' \in \phi_1$	$\begin{cases} b_\ell & ; w' = v' \\ \text{false} & ; \text{otherwise} \end{cases}$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$	$\bigvee_{i=1}^{s} \delta_{v'}(\phi_i)$
$\ell :: \bigvee_{i=1}^{s} \phi_i$	$\bigwedge_{i=1}^{s} \delta_{v'}(\phi_i)$
$\ell :: \exists x \in \phi_1 : \phi_2$	$\delta_{v'}(\phi_2)$
ℓ :: IF ϕ_1 THEN ϕ_2 ELSE ϕ_3	$\delta_{v'}(\phi_2) \wedge \delta_{v'}(\phi_3)$

to v' on every branch of ϕ. We then define

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

We can observe a similar property to the one described in Lemma 1; the only labels that affect whether or not an induced model satisfies a $\delta_{v'}$ formula are the labels of assignment candidates for v'.

Lemma 7. *Let ϕ be an α-TLA$^+$ expression. For any set $A \subseteq \mathcal{L}$ and any variable $v' \in \mathit{Vars}'(\phi)$, it holds that*

$$\mathcal{M}[A] \models \delta_{v'}(\phi) \iff \mathcal{M}[A \cap \text{cand}(v', \phi)] \models \delta_{v'}(\phi)$$

Proof. Analogous to the proof of Lemma 1. □

Our δ transformation is strongly related to boolForm. It turns out, labels necessary to model δ are also necessary to model boolForm. We formalize this in the following way:

Lemma 8. *Let ϕ be and α-TLA$^+$ expression. For any $v' \in Vars'(\phi)$ and $S \subseteq \mathcal{L}$, it holds that if $\mathcal{M}[S] \vDash \delta_{v'}(\phi)$ then $\mathcal{M}[\mathcal{L} \setminus S] \vDash \neg\,\text{boolForm}(\phi)$.*

Proof. We will use induction on the structure of ϕ:

- $\phi = \ell :: \text{FALSE}$: Since $\delta_{v'}(\phi) = \text{false}$ the implication is vacuously true as no model exists.
- $\phi = \ell :: \star(v'_1, \ldots, v'_k)$: Same as for $\phi = \ell :: \text{FALSE}$.
- $\phi = \ell :: w' \in \psi$: If $\delta_{v'}(\phi) = \text{false}$ the implication is vacuously true, since no model exists. If $\delta_{v'}(\phi) = b_\ell$ then $\neg\,\text{boolForm}(\phi) = \neg b_\ell$ and

  $$\mathcal{M}[S] \vDash b_\ell \iff \ell \in S \iff \ell \notin \mathcal{L} \setminus S \iff \mathcal{M}[\mathcal{L} \setminus S] \vDash \neg b_\ell$$

  Thus, the implication holds.
- $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$: Assume as the induction hypothesis, that the lemma holds for each ϕ_i. Let $\mathcal{M}[S] \vDash \delta_{v'}(\phi)$. By definition, $\delta_{v'}(\phi) = \bigvee_{i=1}^{s} \delta_{v'}(\phi_i)$, so we know that there exists a $j \in \{1, \ldots, s\}$, for which $\mathcal{M}[S] \vDash \delta_{v'}(\phi_j)$. By the induction hypothesis, we then know $\mathcal{M}[\mathcal{L} \setminus S] \vDash \neg\,\text{boolForm}(\phi_j)$. Since $\neg\,\text{boolForm}(\phi) = \bigvee_{i=1}^{s} \neg\,\text{boolForm}(\phi_i)$ it also follows that $\mathcal{M}[\mathcal{L} \setminus S] \vDash \neg\,\text{boolForm}(\phi)$, as required.
- $\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$: Analogous to the previous case.
- $\phi = \ell :: \exists x \in \psi \colon \phi_0$: Assume the lemma holds for ϕ_0. It is obvious that, since $\delta_{v'}(\phi) = \delta_{v'}(\phi_0)$ and $\text{boolForm}(\phi) = \text{boolForm}(\phi_0)$, the lemma holds for ϕ as well.
- $\phi = \ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$: Analogous to the disjunction case, since $\text{boolForm}(\phi) = \text{boolForm}(\phi_2 \vee \phi_3)$ and $\delta_{v'}(\phi) = \delta_{v'}(\phi_2 \vee \phi_3)$.

Thus the lemma holds for any α-TLA$^+$ expression ϕ. □

To prove that θ^*_C describes covering sets, we require an additional technical lemma.

Lemma 9. *Let $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$ be an α-TLA$^+$ expression and J a set that intersects every branch of ϕ. Then, J intersects every branch of some ϕ_i non-trivially as well. Formally, take a set $J \subseteq \mathcal{L}$ with the property that*

$$\forall Br \in \text{Branches}(\phi)\ .\ J \cap Br \neq \emptyset$$

Then, the following holds:

$$\exists k \in \{1, \ldots, s\}\ .\ \forall Br \in \text{Branches}(\phi_k)\ .\ J \cap Br \neq \emptyset$$

Proof. We prove this by contradiction. Assume that for every $k \in \{1, \ldots, s\}$ we can find a $Br_k \in \text{Branches}(\phi_k)$ for which $J \cap Br_k = \emptyset$. If we take $Br \coloneqq \bigcup_{k=1}^{s} Br_k$, we generate a branch of ϕ, by Lemma 3. Then, by assumption, $J \cap Br \neq \emptyset$. However, from the way we have defined Br, we see that

$$J \cap Br = J \cap \bigcup_{k=1}^{s} Br_k = \bigcup_{k=1}^{s} (J \cap Br_k) = \bigcup_{k=1}^{s} \emptyset = \emptyset$$

From this contradiction, we deduce that the lemma must hold. □

Using the above lemmas, we can show that the following proposition holds:

Proposition 2. *For every α-TLA$^+$ expression ϕ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C^*(\phi)$ if and only if A is a covering set for ϕ.*

Proof. Firstly, assume $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C^*(\phi)$. This obviously implies that $\mathcal{M}[A] \vDash \theta_H(\phi)$. By Proposition 1, we know A is homogeneous. We will prove that A is a covering set by contradiction:

Take an arbitrary branch $Br \in \text{Branches}(\phi)$ and $v' \in \mathit{Vars}'(\phi)$ and assume that $A \cap Br \cap \text{cand}(v', \phi)$ is empty. Because $\mathcal{M}[A] \vDash \theta_C^*(\phi)$ and $\theta_C^*(\phi) \Rightarrow \delta_{v'}(\phi)$, by definition, it must hold that $\mathcal{M}[A] \vDash \delta_{v'}(\phi)$. By Lemma 7, we know it suffices to consider only the labels from $A \cap \text{cand}(v', \phi)$, which we denote by $A|_{v'}$, for which $\mathcal{M}[A|_{v'}] \vDash \delta_{v'}(\phi)$. By Lemma 8, we can deduce that $\mathcal{M}[\mathcal{L} \setminus A|_{v'}] \vDash \neg\, \text{boolForm}(\phi)$. Since we assumed $Br \cap A|_{v'} = \emptyset$, it follows that $Br \subseteq \mathcal{L} \setminus A|_{v'}$. Because of this we can apply Lemma 2, as $\neg\, \text{boolForm}(\phi)$ in NNF contains only negated atoms, to conclude $\mathcal{M}[Br] \vDash \neg\, \text{boolForm}(\phi)$. However, as Br is a branch it must hold that $\mathcal{M}[Br] \vDash \text{boolForm}(\phi)$ too, which is a contradiction.

Therefore, $Br \cap A|_{v'}$ must be nonempty. As both Br and v' were arbitrary this implies that A is a covering set.

Secondly, consider the opposite direction, where $A \subseteq \mathcal{L}$ is a covering set. We must show that $\mathcal{M}[A] \vDash \theta_C^*(\phi)$, since covering sets are homogeneous, which implies $\mathcal{M}[A] \vDash \theta_H(\phi)$ by Proposition 1. It suffices to see that for every $v' \in \mathit{Vars}'(\phi)$ it holds that $\mathcal{M}[A] \vDash \delta_{v'}(\phi)$. We prove the following statement by induction on the structure of ϕ:

$$\forall v' \in \mathit{Vars}'(\phi)\ .\ [(\forall Br \in \text{Branches}(\phi)\ .\ A|_{v'} \cap Br \neq \emptyset) \Rightarrow \mathcal{M}[A] \vDash \delta_{v'}(\phi)] \tag{6.1}$$

- $\phi = \ell :: \text{FALSE}$: Since $\text{Branches}(\phi) = \{\{\ell\}\}$ and $\ell \notin \text{cand}(\phi)$, no set can satisfy the precondition, so the implication vacuously holds.
- $\phi = \ell :: \star(v'_1, \ldots, v'_k)$: Same as above.
- $\phi = \ell :: w' \in \phi_1$: We know $\text{Branches}(\phi) = \{\{\ell\}\}$. Take an arbitrary $v' \in \mathit{Vars}'(\phi)$ and assume the precondition $\forall Br \in \text{Branches}(\phi)\ .\ A \cap Br \cap \text{cand}(v', \phi) \neq \emptyset$. If $v' \neq w'$ then the precondition generates a contradiction, so (6.1) holds vacuously. Alternatively, if $v' = w'$, we deduce that A must contain $\{\ell\}$. Since $\delta_{w'}(\phi) = b_\ell$, clearly, $\mathcal{M}[A] \vDash b_\ell$.

- $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$: Assume as the induction hypothesis, that (6.1) holds for every $\phi_k, k \in \{1, \ldots, s\}$. Take an arbitrary $v' \in \mathit{Vars}'(\phi)$ and assume the precondition that

$$\forall Br \in \text{Branches}(\phi) \;.\; Br \cap [A \cap \text{cand}(v', \phi)] \neq \emptyset$$

  By applying Lemma 9, with $J = A \cap \text{cand}(v', \phi)$, we can deduce that there is some $k \in \{1, \ldots, s\}$, for which it holds that

$$\forall Br \in \text{Branches}(\phi_k) \;.\; Br \cap [A \cap \text{cand}(v', \phi)] \neq \emptyset$$

  Since any label that is both in Br, which is a branch of ϕ_k, and $\text{cand}(v', \phi)$ is in $\text{cand}(v', \phi_k)$, we see that $B \cap A \cap \text{cand}(v', \phi_k)$ is also nonempty. By the induction hypothesis for ϕ_k, this tells us that $\mathcal{M}[A] \vDash \delta_{v'}(\phi_k)$. Since, by definition, $\delta_{v'}(\phi) = \bigvee_{i=1}^{s} \delta_{v'}(\phi_i)$, it must hold that $\mathcal{M}[A] \vDash \delta_{v'}(\phi)$.

- $\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$: Assume as the induction hypothesis, that (6.1) holds for every $\phi_k, k \in \{1, \ldots, s\}$. Take an arbitrary $v' \in \mathit{Vars}'(\phi)$ and assume the precondition that

$$\forall Br \in \text{Branches}(\phi) \;.\; Br \cap [A \cap \text{cand}(v', \phi)] \neq \emptyset$$

  By applying Lemma 4, we see that $\text{Branches}(\phi) = \bigcup_{i=1}^{s} \text{Branches}(\phi_i)$. We can deduce

$$\forall k \in \{1, \ldots, s\} \;.\; \forall Br \in \text{Branches}(\phi_k) \;.\; Br \cap [A \cap \text{cand}(v', \phi)] \neq \emptyset$$

  By the same argument as in the conjunctive case, any label in $Br \cap \text{cand}(v', \phi)$, where $Br \in \text{Branches}(\phi_k)$, is also in $\text{cand}(v', \phi_k)$, so by using the induction hypothesis, we conclude $M[A] \vDash \delta_{v'}(\phi_k)$ for all $k \in \{1, \ldots, s\}$.

  This means $\mathcal{M}[A] \vDash \bigwedge_{i=1}^{s} \delta_{v'}(\phi_k)$ so as $\delta_{v'}(\phi) = \bigwedge_{i=1}^{s} \delta_{v'}(\phi_k)$ we see that $\mathcal{M}[A] \vDash \delta_{v'}(\phi)$.

- $\phi = \ell :: \exists x \in \phi_1 : \phi_2$: Assume as the induction hypothesis, that (6.1) holds for ϕ_2. Take an arbitrary $v' \in \mathit{Vars}'(\phi)$ and assume the precondition that

$$\forall Br \in \text{Branches}(\phi) \;.\; Br \cap [A \cap \text{cand}(v', \phi)] \neq \emptyset$$

  By applying Lemma 5, we see that $\text{Branches}(\phi) = \text{Branches}(\phi_2)$, so it is clear that $\mathcal{M}[A] \vDash \delta_{v'}(\phi_2)$ by the induction hypothesis. Since $\delta_{v'}(\phi) = \delta_{v'}(\phi_2)$ we know that $\mathcal{M}[A] \vDash \delta_{v'}(\phi)$.

- $\phi = \ell$ :: IF ϕ_1 THEN ϕ_2 ELSE ϕ_3 : Assume as the induction hypothesis, that the statement holds for ϕ_2, ϕ_3. Take an arbitrary $v' \in \mathit{Vars}'(\phi)$ and assume the precondition that

$$\forall Br \in \text{Branches}(\phi) \;.\; Br \cap [A \cap \text{cand}(v', \phi)] \neq \emptyset$$

  By applying Lemma 6, we have $\text{Branches}(\phi) = \text{Branches}(\phi_2) \cup \text{Branches}(\phi_3)$. The rest of this proof is the same as for the disjunctive case, since $\delta_{v'}(\phi) = \delta_{v'}(\phi_2) \wedge \delta_{v'}(\phi_3)$ and $\text{boolForm}(\phi) = \text{boolForm}(\phi_2) \vee \text{boolForm}(\phi_3)$.

So we can conclude, that (6.1) always holds. Take an arbitrary $v' \in \mathit{Vars}'(\phi)$. Since A is a covering set, if also satisfies the precondition of (6.1). Therefore, we know that $\mathcal{M}[A] \vDash \delta_{v'}(\phi)$. As v' was arbitrary, it must be the case that $\mathcal{M}[A] \vDash \theta^*_C(\phi)$. □

With the above lemma we have an encoding of covering sets. However, assignment strategies need to be minimal covering sets, so we need to extend our SMT formula to capture this. It is easy to restrict coverings to the minimal coverings. To do this, we define the set of collocated labels, denoted $\mathrm{Colloc}(\phi)$, as

$$\mathrm{Colloc}(\phi) \coloneqq \{(\ell_1, \ell_2) \in \mathcal{L}^2 \mid \exists Br \in \mathrm{Branches}(\phi) \,.\, \{\ell_1, \ell_2\} \subseteq Br\}$$

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 assignment candidate labels for that variable. We describe these labels by using the sets $\mathrm{Colloc}_{v'}(\phi) \coloneqq \mathrm{Colloc}(\phi) \cap \mathrm{cand}(v', \phi)^2$ and

$$\mathrm{Colloc}_{\mathit{Vars}'}(\phi) \coloneqq \bigcup_{v' \in \mathit{Vars}'(\phi)} \mathrm{Colloc}_{v'}(\phi)$$

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

$$\theta^{\exists!}(\phi) \coloneqq \bigwedge_{\substack{(i,j) \in \mathrm{Colloc}_{\mathit{Vars}'}(\phi) \\ i<j}} \neg(b_i \wedge b_j)$$

We denote by $\theta_C(\phi)$ the formula $\theta^*_C(\phi) \wedge \theta^{\exists!}(\phi)$.

Proposition 3. *For every α-TLA$^+$ expression ϕ and $A \subseteq \mathrm{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C(\phi)$ if and only if A is a minimal covering set for ϕ.*

Proof. Firstly, assume $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C(\phi)$. We already know, from Proposition 2, that such an A is a covering set, since $\mathcal{M}[A] \vDash \theta^*_C(\phi)$ is implied.

Take an arbitrary $Br \in \mathrm{Branches}(\phi)$ and $v' \in \mathit{Vars}'(\phi)$. We must show that $|A \cap Br \cap \mathrm{cand}(v', \phi)| = 1$. We know the set is nonempty, so consider an arbitrary pair $i, j \in A \cap Br \cap \mathrm{cand}(v', \phi)$. Clearly, $\{i, j\} \subseteq Br$ and $\{i, j\} \subseteq \mathrm{cand}(v', \phi)$ so we know $(i, j), (j, i) \in \mathrm{Colloc}_{v'}(\phi)$. We demonstrate that $i = j$ by contradiction.

Assume that $i \neq j$ and w.l.o.g. $i < j$. Since $\mathcal{M}[A] \vDash \theta^{\exists!}(\phi)$ and, by assumption, $i < j$, we must have a term $\neg(b_i \wedge b_j)$, and can conclude $\mathcal{M}[A] \vDash \neg b_i \vee \neg b_j$. However, this is only true if $i \notin A \vee j \notin A$. As we have selected i, j such that $i, j \in A$ we have a contradiction. It then follows that $i = j$ and the intersection is a singleton, as required.

Secondly, assume that a set A is a minimal covering set. In particular, it is also a covering set and thus $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta^*_C(\phi)$, by Proposition 2. To show that $\mathcal{M}[A] \vDash \theta^{\exists!}(\phi)$, take an arbitrary $v' \in \mathit{Vars}'(\phi)$ and $i < j$ for which $(i, j) \in \mathrm{Colloc}_{v'}(\phi)$. We need to see that $\mathcal{M}[A] \vDash \neg(b_i \wedge b_j)$. By definition, there exists a $Br \in \mathrm{Branches}(\phi)$, for which $\{i, j\} \subseteq Br$,

as $\mathrm{Colloc}_{v'}(\phi) \subseteq \mathrm{Colloc}(\phi)$. Since A is a minimal covering set, we know $A \cap Br \cap \mathrm{cand}(v', \phi)$ is a singleton. Both i and j belong to $Br \cap \mathrm{cand}(v', \phi)$ and $i < j$ implies that they are distinct, so one of them must not belong to A. This means $\mathcal{M}[A] \vDash \neg b_i \vee \neg b_j$. As this holds for an arbitrary selection of v' and (i, j), clearly $\mathcal{M}[A] \vDash \theta^{\exists!}(\phi)$, which we needed to show. □

6.5.3 Acyclic assignments

The last step is reasoning about acyclicity. Recall that, for $\ell_1, \ell_2 \in \mathcal{L}$, the relation $\ell_1 \lhd_{v'} \ell_2$ holds if and only if $\ell_1 \in \mathrm{cand}(v', \phi) \wedge v' \in \mathrm{frozen}_\phi(\ell_2)$. It is prudent to see that $\lhd_{v'}$ 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 $\lhd_{v'}$, on all branches.

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

$$\theta^*_A(\phi) \coloneqq \bigwedge_{(i,j) \in \mathrm{Colloc}_\lhd(\phi)} b_i \wedge b_j \Rightarrow R(i) < R(j)$$

where R is an uninterpreted $\mathcal{L} \to \mathbb{N}$ 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 $r \colon \mathbb{N} \to \mathbb{N}$ is the interpretation of R.

A note on injectivity To simplify the analysis, we force R to be injective, when it is restricted to $\mathrm{Labs}(\phi)$. In the strictest sense, this is not necessary. Consider a function R, that satisfies the constraints imposed by $\theta^*_A(\phi)$, but which maps two distinct labels (integers) i, j to the same value, i.e. $R(i) = R(j)$. Note that while i and j are distinct, the pair (i, j) cannot belong to $\mathrm{Colloc}_\lhd(\phi)$. If we define $R'(x) = 2R(x)$ for all $x \neq j$ and $R'(j) = 2R(j) + 1$ we maintain the constraints of $\theta^*_A(\phi)$, but map i and j to different values. To show this, we take an arbitrary label k. If $R(k) < R(j)$, then $R'(k) = 2R(k) < 2R(j) < R'(j)$ and if $R(j) < R(k)$ then $R(j) \leq R(k) - 1$ and $R'(j) = 2R(j) + 1 \leq 2(R(k) - 1) + 1 = 2R(k) - 1 < R'(k)$. By iterating this process for all values where R is not injective, we can obtain (in finitely many steps) an injective function that maintains $\theta^*_A(\phi)$ through every step. However, we opt instead to encode the injectivity of R as an SMT formula, because we can then read the function directly from the solution, without needing to implement this iterative process.

The formula we therefore consider is as follows:

$$\theta_A(\phi) \coloneqq \theta^*_A(\phi) \wedge \bigwedge_{\substack{\ell_1, \ell_j \in \mathrm{Labs}(\phi) \\ \ell_i < \ell_j}} R(\ell_i) \neq R(\ell_j)$$

To prove that θ_A describes acyclic sets, we make use of the following proposition:

Proposition 4. *If $<$ is a strict total order on Y and $f\colon X \to Y$ is injective, then the relation $\prec$ defined by $x_1 \prec x_2 \iff f(x_1) < f(x_2)$ is a strict total order on X.*

Proof. We need to show transitivity, asymmetry, irreflexivity and totality of the relation $\prec$.

transitivity:

$$\begin{aligned} x_1 \prec x_2 \land x_2 \prec x_3 &\iff f(x_1) < f(x_2) \land f(x_2) < f(x_3) \\ &\implies f(x_1) < f(x_3) \\ &\iff x_1 \prec x_3 \end{aligned}$$

asymmetry:

$$\begin{aligned} x_1 \prec x_2 &\iff f(x_1) < f(x_2) \\ &\implies \neg(f(x_2) < f(x_1)) \\ &\iff \neg(x_2 \prec x_1) \end{aligned}$$

irreflexivity:

$$\begin{aligned} \forall y \in Y \,.\, \neg(y < y) &\implies \forall x \in X \,.\, \neg(f(x) < f(x)) \\ &\iff \forall x \in X \,.\, \neg(x \prec x) \end{aligned}$$

totality:

$$\forall y_1, y_2 \in Y \,.\, y_1 < y_2 \lor y_2 < y_1 \lor y_1 = y_2$$

implies

$$\forall x_1, x_2 \in X \,.\, f(x_1) < f(x_2) \lor f(x_2) < f(x_1) \lor f(x_1) = f(x_2)$$

which is equivalent to

$$\forall x_1, x_2 \in X \,.\, x_1 \prec x_2 \lor x_2 \prec x_1 \lor x_1 = x_2$$

for injective f.

Thus $\prec$ is a strict total order on X □

With Proposition 4, we can show that the following porposition holds.

Proposition 5. *For every α-TLA$^+$ expression ϕ and $A \subseteq \mathrm{Labs}(\phi)$, there is a function $r\colon \mathbb{N} \to \mathbb{N}$, for which $(\mathcal{M}[A], r) \vDash \theta_H(\phi) \land \theta_A(\phi)$ if and only if A is acyclic.*

Proof. Firstly, assume that there exists an $r\colon \mathbb{N} \to \mathbb{N}$, for which $(\mathcal{M}[A], r) \vDash \theta_H(\phi) \wedge \theta_A(\phi)$. This obviously implies that $\mathcal{M}[A] \vDash \theta_H(\phi)$. By Proposition 1, we know A is homogeneous. We define $\prec_A$ using r:

$$\ell_1 \prec_A \ell_2 \iff r(\ell_1) < r(\ell_2)$$

Clearly, $<$ is a strict total order on $\mathbb{N}$. We have ensured that

$$\bigwedge_{\substack{\ell_1,\ell_j \in \text{Labs}(\phi) \\ \ell_i < \ell_j}} r(\ell_i) \neq r(\ell_j)$$

so r restricted to $\text{Labs}(\phi)$ is injective. As $A \subseteq \text{Labs}(\phi)$ we know that r restricted to A is injective as well. We can then use Proposition 4, for the function $f = r|_A\colon A \to \mathbb{N}$, to conclude that such a $\prec_A$ is a strict total order on A. Now take an arbitrary branch $Br \in \text{Branches}(\phi)$, two labels $\ell_1, \ell_2 \in A \cap Br$ and a variable $v' \in \mathit{Vars}'(\phi)$. If the relation $\ell_1 \lhd_{v'} \ell_2$ does not hold, the implication $\ell_1 \lhd_{v'} \ell_2 \Rightarrow \ell_1 \prec_A \ell_2$ is vacuously correct. If it does, since ℓ_1, ℓ_2 belong to $A \cap Br$, we know that $(\ell_1, \ell_2) \in \text{Colloc}_\lhd(\phi)$. As $(\mathcal{M}[A], r) \vDash \theta_A(\phi)$ it is also the case that $(\mathcal{M}[A], r) \vDash \theta^*_A(\phi)$. We know that $\mathcal{M}[A] \vDash b_{\ell_1} \wedge b_{\ell_2}$ so it must be the case that $r(\ell_1) < r(\ell_2)$. But then, by definition, $\ell_1 \prec_A \ell_2$. Because Br, ℓ_1, ℓ_2 and v' were arbitrary, we can conclude that A is acyclic.

Secondly, assume A is acyclic. We must show that $\mathcal{M}[A] \vDash \theta^*_A(\phi)$, since acyclic sets are homogeneous, which implies $\mathcal{M}[A] \vDash \theta_H(\phi)$ by Proposition 1. We can take the strict total order $\prec_A$ and arbitrarily extend it to a strict total order $\prec$ on $\text{Labs}(\phi)$. Because of this, there exists an ordering function $\text{ord}\colon \text{Labs}(\phi) \to \{1, \ldots, |\text{Labs}(\phi)|\}$ with the property

$$\ell_1 \prec \ell_2 \iff \text{ord}(\ell_1) < \text{ord}(\ell_2)$$

we can define $r\colon \mathbb{N} \to \mathbb{N}$ as

$$r(n) = \begin{cases} \text{ord}(n) & ; n \in \text{Labs}(\phi) \\ 1 & ; \text{otherwise} \end{cases}$$

Let us first see that $(\mathcal{M}[A], r) \vDash \theta^*_A(\phi)$. Take an arbitrary pair $(i, j) \in \text{Colloc}_\lhd(\phi)$. We need to show that $(\mathcal{M}[A], r) \vDash b_i \wedge b_j \Rightarrow R(i) < R(j)$. If $i \notin A$ or $j \notin A$ then $b_i \wedge b_j$ evaluates to false and the implication in satisfied. If both i and j belong to A, then we take an arbitrary $Br \in \text{Branches}(\phi)$ containing both of them (it exists, since $(i, j) \in \text{Colloc}(\phi)$ as $\text{Colloc}_\lhd(\phi) \subseteq \text{Colloc}(\phi)$) and the variable $v' \in V$ for which $i \lhd_{v'} j$. As A is acyclic, we can instantiate the acyclicity criterion for our choice of Br, i, j and v' and conclude $i \prec_A j$. Because $\prec$ extends $\prec_A$ it must be the case that $\text{ord}(i) < \text{ord}(j)$ and, because $r|_{\text{Labs}(\phi)} = \text{ord}$, also $r(i) < r(j)$. So $(\mathcal{M}[A], r)$ models $\theta^*_A(\phi)$. We conclude the proof by showing that this r also models the formula

$$\bigwedge_{\substack{\ell_1,\ell_j \in \text{Labs}(\phi) \\ \ell_i < \ell_j}} R(\ell_i) \neq R(\ell_j)$$

If $\ell_1, \ell_2 \in \text{Labs}(\phi)$ then $r(\ell_1) = \text{ord}(\ell_1)$ and $r(\ell_2) = \text{ord}(\ell_2)$. It then follows, as ord is bijective, that either $r(\ell_1) < r(\ell_2)$ or vice-versa. In any case, $r(\ell_1) \neq r(\ell_2)$. Altogether, this implies $(\mathcal{M}[A], r) \vDash \theta_A(\phi)$. □

Thus, we can prove the previously foreshadowed theorem.

Theorem 1. *For every α-TLA$^+$ formula ϕ and $A \subseteq \text{Labs}(\phi)$, it holds that $\mathcal{M}[A] \vDash \theta(\phi)$ if and only if A is an assignment strategy for ϕ.*

Proof. Let ϕ be an α-TLA$^+$ formula and $A \subseteq \text{Labs}(\phi)$. By definition, $\theta(\phi) = \theta_H(\phi) \wedge \theta_C(\phi) \wedge \theta_A(\phi)$.

Firstly, assume $\mathcal{M}[A] \vDash \theta(\phi)$. As $\theta(\phi)$ implies both $\theta_H(\phi) \wedge \theta_C(\phi)$ and $\theta_H(\phi) \wedge \theta_A(\phi)$, we know A is a minimal covering and acyclic, by propositions 3 and 5 respectively. By definition, this means A is an assignment strategy.

Secondly, assume A is an assignment strategy. In particular, A a minimal covering, so $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C(\phi)$. Similarly, as A is acyclic, we know that $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_A(\phi)$. It therefore follows that $\mathcal{M}[A] \vDash \theta_H(\phi) \wedge \theta_C(\phi) \wedge \theta_A(\phi)$, that is, $\mathcal{M}[A] \vDash \theta(\phi)$. □

6.6 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. We will later define symbolic transitions as special kinds of slices.

In TLA$^+$, there are two kinds of variables: rigid variables that correspond to the variables declared with 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 [Mer08a]. An interpretation $\mathcal{I}$ defines a universe of $|\mathcal{I}|$ 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 ξ is a mapping from rigid variables to values. Given an interpretation $\mathcal{I}$, a pair of states (s, s'), and a valuation ξ, 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', ξ. With these definitions, $M = (\mathcal{I}, \xi, s, s')$ is a model for E, if E is equivalent to true under M.

6.6.1 Slices

Let ϕ be a formula and $S \subseteq \mathcal{L}$. We define ϕ *sliced by* S, denoted slice(ϕ, S) in Table 6.6. Intuitively, slices contain only those terminal expressions (stars and assignemnt candidates) whose labels appear in S.

Table 6.6: The definition of slice(ϕ, S)

α-**TLA$^+$ formula** ϕ	slice(ϕ, S)
ℓ :: FALSE	ℓ :: FALSE
$\ell :: \star(v'_1, \ldots, v'_1)$ or $\ell :: v' \in \phi_1$	$\begin{cases} \phi & ; \ell \in S \\ \ell :: \text{FALSE} & ; \text{otherwise} \end{cases}$
$\ell :: \bigwedge_{i=1}^{s} \phi_i$	$\ell :: \bigwedge_{i=1}^{s} \text{slice}(\phi_i, S)$
$\ell :: \bigvee_{i=1}^{s} \phi_i$	$\ell :: \bigvee_{i=1}^{s} \text{slice}(\phi_i, S)$
$\ell :: \exists x \in \phi_1 : \phi_2$	$\ell :: \exists x \in \phi_1 : \text{slice}(\phi_2, S)$
ℓ :: IF ϕ_1 THEN ϕ_2 ELSE ϕ_3	ℓ :: IF ϕ_1 THEN slice(ϕ_2, S) ELSE slice(ϕ_3, S)

We can see how slice(ϕ, A), like boolForm, depends only on labels of ϕ.

Lemma 10. *Let ϕ be an α-TLA$^+$ expression. For any set $A \subseteq \mathcal{L}$, it holds that*

$$\text{slice}(\phi, A) = \text{slice}(\phi, A \cap \text{Labs}(\phi))$$

Proof. Analogous to the proof of Lemma 1. □

Below, we show that the branches and their induced slices naturally decompose a TLA$^+$ formula. Let ϕ be an α-TLA$^+$ expression and $\gamma(\phi)$ its corresponding TLA$^+$formula. Then, the following propositions hold:

Proposition 6. *Let ϕ be an α-TLA$^+$ expression and $M = (\mathcal{I}, \xi, s, s')$ a model of the TLA$^+$ formula $\gamma(\phi)$. There exists a branch Br of ϕ such that M is also a model of $\gamma(\text{slice}(\phi, Br))$.*

Proof. Assume that $M = (\mathcal{I}, \xi, s, s')$ is a model of $\gamma(\phi)$. We prove the proposition by induction on the structure of ϕ:

$\phi = \ell$ :: FALSE:

Since $\gamma(\phi)$ is FALSE it cannot have a model. Because of the contradiction, the proposition vacuously holds.

$\phi = \ell :: \star(v'_1, \ldots, v'_k)$:

The only branch is exactly $Br = \{\ell\}$. Then slice$(\phi, Br) = \phi$ so $\gamma(\text{slice}(\phi, Br)) = \gamma(\phi)$, therefore M is clearly a model of $\gamma(\text{slice}(\phi, Br))$.

$\phi = \ell :: v' \in \hat{\ell} :: \star(v'_1, \ldots, v'_k)$:

Analogous to the previous case.

$\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$:

Assume as the induction hypothesis, that the proposition holds for every ϕ_i, where $i \in \{1, \ldots, s\}$.

Since $\gamma(\phi) = \bigwedge_{i=1}^{s} \gamma(\phi_i)$ and M is a model of $\gamma(\phi)$, it must also be a model for each $\gamma(\phi_i)$. By the induction hypothesis we can deduce that there are branches $Br_1 \in \text{Branches}(\phi_1), \ldots, Br_s \in \text{Branches}(\phi_s)$, for each subformula, such that M is a model for $\gamma(\text{slice}(\phi_i, Br_i))$, for each $i \in \{1, \ldots, s\}$.

By Lemma 3, we know that $Br \coloneqq \bigcup_{i=1}^{s} Br_i$ is a branch of ϕ. We need to show that M is a model for $\gamma(\text{slice}(\phi, Br))$. It suffices to show that M is a model for each $\gamma(\text{slice}(\phi_i, Br))$, since $\text{slice}(\phi, Br) = \bigwedge_{i=1}^{s} \text{slice}(\phi_i, Br)$, by definition.

Let us pick an arbitrary $i \in \{1, \ldots, s\}$. Applying Lemma 10, we know that $\text{slice}(\phi_i, Br) = \text{slice}(\phi_i, Br \cap \text{Labs}(\phi_i))$. As often before, we conclude that $Br \cap \text{Labs}(\phi_i) = Br_i$. We know M is a model for $\gamma(\text{slice}(\phi_i, Br_i))$, so it follows that it is also a model for $\gamma(\text{slice}(\phi_i, Br))$.

$\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$:

Assume as the induction hypothesis, that the proposition holds for every ϕ_i, where $i \in \{1, \ldots, s\}$.

Since $\gamma(\phi) = \bigvee_{i=1}^{s} \gamma(\phi_i)$ and M is a model of $\gamma(\phi)$, there must exist an index $i \in \{1, \ldots, s\}$, such that M is a model for $\gamma(\phi_i)$. By the induction hypothesis we can deduce that there is a branch $Br \in \text{Branches}(\phi_i)$, such that M is a model for $\gamma(\text{slice}(\phi_i, Br_i))$.

By Lemma 4, we know that Br is also a branch of ϕ. We need to show that M is a model for $\gamma(\text{slice}(\phi, Br))$. We already know M is a model for $\gamma(\text{slice}(\phi_i, Br))$, and since $\text{slice}(\phi, Br) = \bigvee_{i=1}^{s} \text{slice}(\phi_i, Br)$, by definition, M must also be a model for $\gamma(\text{slice}(\phi, Br))$.

$\phi = \ell :: \exists x \in \psi \colon \phi_0$:

Assume as the induction hypothesis, that the proposition holds for ϕ_0.

Since, by assumption, M is a model of $\gamma(\phi)$, we know there exists an $x_0 \in \gamma(\psi)$ for which M is a model for $\gamma(\phi_0)[x_0/x]$, or alternatively, $M' = (\mathcal{I}, \xi[x \mapsto x_0], s, s')$ is a model for $\gamma(\phi_0)$. Using the induction hypothesis, there is a branch Br of ϕ_0, for which M' is a model for $\gamma(\text{slice}(\phi_0, Br))$. By Lemma 5, we know Br is also a branch for ϕ, and by definition, we know $\text{slice}(\phi, Br) = \exists x \in \psi \colon \text{slice}(\phi_0, Br)$.

As M' is a model for $\gamma(\text{slice}(\phi_0, Br))$, we can equivalently say that M is a model for $\gamma(\text{slice}(\phi_0, Br))[x_0/x]$. This must mean that M is also a model for $\exists x \in \gamma(\psi) \colon \gamma(\text{slice}(\phi_0, Br))$, which is exactly $\gamma(\phi)$.

$\phi = \ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$:

Assume as the induction hypothesis, that the proposition holds for ϕ_2, ϕ_3.

As M is a model for $\gamma(\phi)$, we know that either $\gamma(\phi_1)$ is true under M, and M is a model of $\gamma(\phi_2)$, or $\gamma(\phi_1)$ is false under M, and M is a model of $\gamma(\phi_3)$. In either case, if M is a model of $\gamma(\phi_i)$, we can use the induction hypothesis, to deduce that there exists a branch $Br \in \text{Branches}(\phi_i)$, for which M is a model of $\gamma(\text{slice}(\phi_i, Br))$. By Lemma 6, this branch is also a branch of ϕ.

By definition, $\text{slice}(\phi, Br) = \text{IF } \phi_1 \text{ THEN } \text{slice}(\phi_2, Br) \text{ ELSE } \text{slice}(\phi_3, Br)$ and therefore $\gamma(\text{slice}(\phi, Br)) = \text{IF } \gamma(\phi_1) \text{ THEN } \gamma(\text{slice}(\phi_2, Br)) \text{ ELSE } \gamma(\text{slice}(\phi_3, Br))$.

We know that if $\gamma(\phi_1)$ is true, M is a model of $\gamma(\text{slice}(\phi_2, Br))$, otherwise M is a model of $\gamma(\text{slice}(\phi_3, Br))$, so M is a model of $\gamma(\text{slice}(\phi, Br))$.

□

Proposition 7. *Let ϕ be an α-TLA$^+$ expression, Br a branch of ϕ, and $M = (\mathcal{I}, \xi, s, s')$ a model of the TLA$^+$ formula $\gamma(\text{slice}(\phi, Br))$. Then, M is also a model of $\gamma(\phi)$.*

Proof. Assume that Br is a branch of ϕ, and $M = (\mathcal{I}, \xi, s, s')$ models $\gamma(\text{slice}(\phi, Br))$. We prove the proposition by induction on the structure of ϕ:

$\phi = \ell :: \text{FALSE}$:

Since $\gamma(\text{slice}(\phi, S))$ is FALSE for any S, it cannot have a model. Because of the contradiction, the proposition vacuously holds.

$\phi = \ell :: \star(v'_1, \ldots, v'_k)$:

The only branch is exactly $Br = \{\ell\}$. Then $\text{slice}(\phi, Br) = \phi$ so $\gamma(\text{slice}(\phi, Br)) = \gamma(\phi)$, therefore M is clearly a model of $\gamma(\phi)$.

$\phi = \ell :: v' \in \hat{\ell} :: \star(v'_1, \ldots, v'_k)$:

Analogous to the previous case.

$\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$:

Assume as the induction hypothesis, that the proposition holds for every ϕ_i, where $i \in \{1, \ldots, s\}$.

By definition, $\text{slice}(\phi, Br) = \bigwedge_{i=1}^{s} \text{slice}(\phi_i, Br)$. Applying Lemma 10 to each subexpression, and Lemma 3, we see that $\text{slice}(\phi, Br) = \bigwedge_{i=1}^{s} \text{slice}(\phi_i, Br_i)$, where each Br_i is a branch of ϕ_i.

The above implies that M is a model for each $\gamma(\text{slice}(\phi_i, Br_i))$. By the induction hypothesis, M is then also a model for $\gamma(\phi_i)$, from which it must follow that M is a model for $\gamma(\phi)$.

$\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$:

Assume as the induction hypothesis, that the proposition holds for every ϕ_i, where $i \in \{1, \ldots, s\}$.

By definition, $\text{slice}(\phi, Br) = \bigvee_{i=1}^{s} \text{slice}(\phi_i, Br)$, so there is an $i \in \{1, \ldots, s\}$, for which M is a model of $\gamma(\text{slice}(\phi_i, Br))$. It must be the case that Br is also a branch for ϕ_i. We show this by means of contradiction: Each branch of ϕ is a branch or one ϕ_j by Lemma 4. Assume $Br \in \text{Branches}(\phi_j)$ and $i \neq j$. Because the label sets are pairwise disjoint, Lemma 10 helps us see that $\text{slice}(\phi_i, Br) = \text{slice}(\phi_i, Br \cap \text{Labs}(\phi_i)) = \text{slice}(\phi_i, \emptyset)$. But then $\gamma(\text{slice}(\phi_i, \emptyset))$ cannot have a model, which contradicts our assumptions (slicing by the empty set always returns a formula equivalent to FALSE). We conclude that Br is indeed a branch of ϕ_i.

This allows us to use the induction hypothesis, to show that M is a model of $\gamma(\phi_i)$. It trivially follows that M is also a model of $\bigvee_{j=1}^{s} \gamma(\phi_j) = \gamma(\phi)$.

$\phi = \ell :: \exists x \in \psi \colon \phi_0$:

Assume as the induction hypothesis, that the proposition holds for ϕ_0.

By Lemma 5, we know that Br is a branch of ϕ_0. As we know that $\text{slice}(\phi, Br) = \exists x \in \psi \colon \text{slice}(\phi_0, Br)$, and $\gamma(\text{slice}(\phi, Br)) = \exists x \in \gamma(\psi) \colon \gamma(\text{slice}(\phi_0, Br))$, we can use the fact that M is a model for $\gamma(\text{slice}(\phi, Br))$ to determine that there exists an $x_0 \in \gamma(\psi)$, for which M is a model for $\gamma(\text{slice}(\phi_0, Br))[x_0/x]$. This is equivalent to saying that $M' = (\mathcal{I}, \xi[x \mapsto x_0], s, s')$ is a model for $\gamma(\text{slice}(\phi_0, Br))$.

We then apply the induction hypothesis and deduce that M' is a model for $\gamma(\phi_0)$, or equivalently, that M is a model for $\gamma(\phi_0)[x_0/x]$. It follows that M is a model for $\exists x \in \gamma(\psi) \colon \gamma(\phi_0) = \gamma(\phi)$.

$\phi = \ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$:

Assume as the induction hypothesis, that the proposition holds for ϕ_2, ϕ_3.

By definition, $\text{slice}(\phi, Br) = \text{IF } \phi_1 \text{ THEN } \text{slice}(\phi_2, Br) \text{ ELSE } \text{slice}(\phi_3, Br)$ and $\gamma(\text{slice}(\phi, Br)) = \text{IF } \gamma(\phi_1) \text{ THEN } \gamma(\text{slice}(\phi_2, Br)) \text{ ELSE } \gamma(\text{slice}(\phi_3, Br))$. Suppose $\gamma(\phi_1)$ is true under M. Then M must satisfy $\gamma(\text{slice}(\phi_2, Br))$, otherwise it must satisfy $\gamma(\text{slice}(\phi_3, Br))$. We need to show that Br must be a branch of ϕ_2, resp. ϕ_3. This proof is analogous to the disjunctive case; we leverage Lemma 6 and Lemma 10. Using the induction hypothesis, we see that M must therefore be a model for $\text{IF } \gamma(\phi_1) \text{ THEN } \gamma(\phi_2) \text{ ELSE } \gamma(\phi_3) = \gamma(\phi)$.

□

With these two propositions, we have shown that a TLA$^+$ formula has a model if and only if one of its slices has a model. However, the number of branches to check may still be quite large. In the remainder of this section we demonstrate that there is a coarser decomposition of labels, the slices of which also decompose the original TLA$^+$ formula.

Firstly, we show that enlarging a slicing set preserves models. Formally:

Proposition 8. *Let ϕ be an α-TLA$^+$ expression. For every $S, T \subseteq \text{Labs}(\phi)$, every model M of TLA$^+$ formula $\gamma(\text{slice}(\phi, S))$ is also a model of $\gamma(\text{slice}(\phi, S \cup T))$.*

Proof. Assume that M is a model for $\gamma\left(\text{slice}(\phi, S)\right)$ and $T \subseteq \text{Labs}(\phi)$. If $S = \emptyset$, we have a contradiction, since $\gamma(\text{slice}(\phi, \emptyset))$ is equivalent to FALSE, which has no model, so any conclusion follows. If $T = \emptyset$, the result is trivial. Assume henceforth that $S, T \neq \emptyset$.

We prove the proposition by induction on the structure of ϕ:

$\phi = \ell :: \text{FALSE}$:

Since $\text{Labs}(\phi) = \{\ell\}$, it must be the case that $S = T = \{\ell\}$, so the conclusion trivially follows.

$\phi = \ell :: \star\left(v'_1, \ldots, v'_k\right)$:

Analogous to the previous case.

$\phi = \ell :: v' \in \hat{\ell} :: \star\left(v'_1, \ldots, v'_k\right)$:

As $\text{Labs}(\phi) = \{\ell, \hat{\ell}\}$, S must contain ℓ, as otherwise $\gamma(\text{slice}(\phi, S))$ has no model. The only new case is when $S \cup T = \{\ell, \hat{\ell}\}$. Even then, $\text{slice}(\phi, S \cup T) = \text{slice}(\phi, S)$ and M is a model of $\gamma(\text{slice}(\phi, S \cup T))$.

$\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$:

Assume as the induction hypothesis, that the proposition holds for every ϕ_i, where $i \in \{1, \ldots, s\}$.

Since M is a model for $\gamma(\text{slice}(\phi, S))$, it must be a model for each $\gamma(\text{slice}(\phi_i, S))$. Using the induction hypothesis, we can conclude that M is a model for each $\gamma(\text{slice}(\phi_i, S \cup T))$, so it is also a model for $\bigwedge_{i=1}^{s} \gamma(\text{slice}(\phi_i, S \cup T)) = \gamma(\text{slice}(\phi, S \cup T))$.

$\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$:

Analogous to the previous case.

$\phi = \ell :: \exists x \in \psi \colon \phi_0$:

Assume as the induction hypothesis, that the proposition holds for ϕ_0.

Since M is a model for $\gamma(\text{slice}(\phi, S))$, it must be a model for $\gamma(\text{slice}(\phi_0, S))$ and therefore, by the induction hypothesis, also a model for $\gamma(\text{slice}(\phi_0, S \cup T)) = \gamma(\text{slice}(\phi, S \cup T))$.

$\phi = \ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3$:

Analogous to the disjunctive case.

□

It is easy to see that converse of Proposition 8 does not hold. For instance, take the empty set as S and $\text{Labs}(\phi)$ as T. This implies the following:

$$\gamma(\text{slice}(\phi, S)) = \text{FALSE} \text{ and } \text{slice}(\phi, S \cup T) = \phi.$$

Obviously, FALSE cannot have a model, regardless of whether $\gamma(\phi)$ has one or not.

Due to Propositions 6 and 7, it would suffice to consider the set Branches(ϕ), together with an assignment strategy A, to generate transitions. However, it is often the case that, for two distinct branches Br_1 and Br_2, the same assignments in A are chosen, that is, the intersections $Br_1 \cap A$ and $Br_2 \cap A$ are the same. We show that one can reduce the number of considered transitions, by analyzing how various branches intersect A. Instead of separately analyzing slice(ϕ, Br_1) and slice(ϕ, Br_2) for such branches, we will be able to consider just slice($\phi, Br_1 \cup Br_2$).

6.6.2 Symbolic transitions

An assignment strategy A naturally defines an equivalence relation $\sim_A$ on Branches(ϕ), given by $Br_1 \sim_A Br_2$ if and only if $Br_1 \cap A = Br_2 \cap A$. We use the notation $[Br]_A$ to refer to the equivalence class of Br by $\sim_A$, that is, the set $\{X \in \text{Branches}(\phi) \mid Br \sim_A X\}$.

Definition 5. *Let ϕ be an α-TLA$^+$ expression, A an assignment strategy for ϕ and Br a branch of ϕ. Using $X = [Br]_A$ and $Y = \bigcup_{Z \in X} Z$, we define the symbolic transition generated by Br and A to be* slice(ϕ, Y).

Example 3. *Let us look the example on page 43 again. The formula ϕ has two assignment strategies $A_1 = \{\ell_2\}$, and $A_2 = \{\ell_4, \ell_5\}$. If the first assignment strategy A_1 is chosen, we have that $Br_1 \cap A_1 = Br_2 \cap A_1 = \{\ell_2\}$. This implies that Br_1 and Br_2 are in the same equivalence class, or $Br_1 \sim_{A_1} Br_2$. Therefore, we have only one symbolic transition which is exactly ϕ. However, if A_2 is selected, branches Br_1 and Br_2 are not equivalent because $Br_1 \cap A_2 = \{\ell_4\}$ and $Br_2 \cap A_2 = \{\ell_5\}$. Therefore, we have two symbolic transitions:*

$$T_1 = \ell_1 :: [[\ell_2 :: x' \in \star] \wedge [\ell_3 :: [[\ell_4 :: x' \in \star] \vee \ell_5 :: \text{FALSE}]]]$$
$$T_2 = \ell_1 :: [[\ell_2 :: x' \in \star] \wedge [\ell_3 :: [\ell_4 :: \text{FALSE} \vee [\ell_5 :: x' \in \star]]]]$$

The first assignment strategy A_1 seems to be better than A_2 because A_1 generates fewer symbolic transitions than A_2. 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 $\sim_A$ allows us to use Proposition 8 in the following interesting way:

Proposition 9. *Let ϕ be an α-TLA$^+$ expression. For any selection $Br_1, \ldots, Br_k$ from the branches of ϕ, the following holds: If there exists a model $M = (\mathcal{I}, \xi, s, s')$ of the formula $\gamma(\text{slice}(\phi, Br_1 \cup \cdots \cup Br_k))$, then M must be a model of $\gamma(\text{slice}(\phi, Br))$, for some branch $Br \in$ Branches(ϕ). Additionally, if there is an assignment strategy A for ϕ, such that $Br_1, \ldots, Br_k$ all belong to the same equivalence class $[B]_A$, then M must be a model of $\gamma(\text{slice}(\phi, Br))$, for some branch $Br \in [B]_A$.*

Proof. Denote by S the union $Br_1 \cup \cdots \cup Br_k$. We prove the proposition by induction on the structure of ϕ:

- $\phi = \ell :: \text{FALSE}$:

  Since, for any $T \subseteq \text{Labs}(\phi)$ the formula $\gamma(\text{slice}(\phi, T))$ is equivalent to FALSE, it does not have a model, so the implication vacuously holds.

- $\phi = \ell :: \star(v'_1, \ldots, v'_k)$:

  Assume there exists a model M of $\gamma(\text{slice}(\phi, S))$. This means that ℓ must belong to S, otherwise $\text{slice}(\phi, S)$ is $\ell :: \text{FALSE}$ and γ applied to $\ell :: \text{FALSE}$ is FALSE, which does not have a model. As ϕ has exactly one branch, the rest of the proposition follows trivially, since $S \supseteq Br_1$. Consequently, $\text{slice}(\phi, Br_1)$, ϕ and $\text{slice}(\phi, S)$ are all the same expression, by Lemma 10. As M is a model of $\gamma(\text{slice}(\phi, S))$ it is also a model of $\text{slice}(\phi, Br_1)$. It is clear that if all chosen branches belong to $[B]_A$, then, in particular, $Br_1 \in [B]_A$.

- $\phi = \ell :: w' \in \phi_1$:

  Analogous to the previous case.

- $\phi = \ell :: \bigwedge_{i=1}^{s} \phi_i$:

  Assume, as the induction hypothesis, that the proposition holds for all ϕ_i.

  By definition, $\text{slice}(\phi, S) = \bigwedge_{i=1}^{s} \text{slice}(\phi_i, S)$. Assume, that M is a model of $\gamma(\text{slice}(\phi, S))$. Then, $\gamma(\text{slice}(\phi, S)) = \bigwedge_{i=1}^{s} \gamma(\text{slice}(\phi_i, S))$ and M is a model of $\gamma(\text{slice}(\phi_i, S))$ for every i. By Lemma 3, we know that for each $i = 1, \ldots, k$ there exist branches $Br_i^1, \ldots, Br_i^s$ of $\phi_1, \ldots, \phi_s$, such that

  $$Br_i = \bigcup_{j=1}^{s} Br_i^j$$

  Take an arbitrary $i \in \{1, \ldots, s\}$. By Lemma 10, we know that

  $$\text{slice}(\phi_i, S) = \text{slice}(\phi_i, S \cap \text{Labs}(\phi_i))$$

  We can see the following:

  $$\begin{aligned} S \cap \text{Labs}(\phi_i) &= \left(\bigcup_{t=1}^{k} Br_t \right) \cap \text{Labs}(\phi_i) \\ &= \left(\bigcup_{t=1}^{k} \bigcup_{j=1}^{s} Br_t^j \right) \cap \text{Labs}(\phi_i) \\ &= \bigcup_{t=1}^{k} \bigcup_{j=1}^{s} \left(Br_t^j \cap \text{Labs}(\phi_i) \right) \end{aligned}$$

As each Br_t^j is a branch of ϕ_j, all of the intersections are either Br_t^j, if $i = j$, or empty. Consequently:

$$S \cap \text{Labs}(\phi_i) = \bigcup_{t=1}^{k} Br_t^i$$

By design, Br_t^i is a branch of ϕ_i, for each t. It means we can use our induction hypothesis for ϕ_i to deduce that there must exist a $Br^i \in \text{Branches}(\phi_i)$, for which M is a model of $\gamma(\text{slice}(\phi_i, Br^i))$. As i was arbitrary, this holds for every selection of i. We thus obtain a collection of branches, $Br^1, \ldots, Br^s$, for which it holds that M is a model of $\gamma(\text{slice}(\phi_i, Br^i))$ for every $i \in \{1, \ldots, s\}$. Using $Br_0 = \bigcup_{i=1}^{s} Br^i$ and Proposition 8, we deduce that M is a model of $\gamma(\text{slice}(\phi_i, Br_0))$, for each $i \in \{1, \ldots, s\}$. By Lemma 3, Br_0 is a branch of ϕ. So it follows that M is a model of $\gamma(\text{slice}(\phi, Br_0))$.

Assume additionally that $Br_1, \ldots, Br_k \in [B]_A$ for some assignment strategy A and some equivalence class $[B]_A$ of $\sim_A$.

By definition, $Br_i \cap A = Br_1 \cap A$ for all $i \in \{1, \ldots, k\}$. It follows, that $(Br_i \cap \text{Labs}(\phi_j)) \cap A = (Br_1 \cap \text{Labs}(\phi_j)) \cap A$ for all $i \in \{1, \ldots, k\}$ and all $j \in \{1, \ldots s\}$. This means that the sets $Br_1^i, \ldots, Br_k^i$ are equivalent for all $i \in \{1, \ldots, s\}$, since $Br_i \cap \text{Labs}(\phi_j) = Br_i^j$. By the induction hypothesis, this implies that Br^i is equivalent to Br_1^i, for all $i \in \{1, \ldots, s\}$. Altogether:

$$\begin{aligned} Br_0 \cap A &= \bigcup_{i=1}^{s} (Br^i \cap A) \\ &= \left(\bigcup_{i=1}^{s} Br_1^i \cap A \right) \\ &= \left(\bigcup_{i=1}^{s} Br_1^i \right) \cap A \\ &= Br_1 \cap A \end{aligned}$$

Thus we see that Br_0 is equivalent to Br_1 and, by transitivity, to all of the branches $Br_1, \ldots, Br_k$.

- $\phi = \ell :: \bigvee_{i=1}^{s} \phi_i$:

  Assume, as the induction hypothesis, that the proposition holds for all ϕ_i.

  By definition, $\text{slice}(\phi, S) = \bigvee_{i=1}^{s} \text{slice}(\phi_i, S)$. It follows that $\gamma(\text{slice}(\phi, S)) = \bigvee_{i=1}^{s} \gamma(\text{slice}(\phi_i, S))$. If we assume that M is a model of $\gamma(\text{slice}(\phi, S))$, there must exist an $i \in \{1, \ldots, k\}$, for which M is a model of $\gamma(\text{slice}(\phi_i, S))$ By Lemma 10, we know that

  $$\text{slice}(\phi_i, S) = \text{slice}(\phi_i, S \cap \text{Labs}(\phi_i))$$

Additionally, Lemma 4 guarantees that for each $j = 1, \ldots, k$ the set $Br_j \cap \text{Labs}(\phi_i)$ is either Br_j or empty. Because $\gamma(\text{slice}(\phi_i, S \cap \text{Labs}(\phi_i)))$ has a model, the set $S \cap \text{Labs}(\phi_i)$ is not empty. It is therefore a union of $0 < l \leq k$ branches $Br'_1, \ldots, Br'_l$ from $\text{Branches}(\phi_i)$.

By the induction hypothesis for ϕ_i, there exists a $Br^i \in \text{Branches}(\phi_i)$, for which M is a model of $\gamma(\text{slice}(\phi_i, Br^i))$. By Lemma 4, ^{i} is also a branch for ϕ.

Assume additionally that $Br_1, \ldots, Br_k \in [B]_A$ for some assignment strategy A and some equivalence class $[B]_A$ of $\sim_A$. Trivially, all branches $Br'_1, \ldots, Br'_l$ are equivalent as well, since each of them corresponds to some Br_j. Therefore, $Br^i \sim_A Br'_1$. As Br'_1 is equal to one of the branches $Br_1, \ldots, Br_k$, which are all equivalent, we see that Br^i is equivalent to Br_1 and, by transitivity, to all of the branches $Br_1, \ldots, Br_k$.

- $\phi = \ell :: \exists x \in \phi_1 \colon \phi_2 :$

  Assume, as the induction hypothesis, that the proposition holds for ϕ_2. By definition, $\text{slice}(\phi, S) = \exists x \in \phi_1 \colon \text{slice}(\phi_2, S)$. Assume, that M is a model of $\gamma(\text{slice}(\phi, S))$. Then,

  $$\gamma(\text{slice}(\phi, S)) = \ell :: \exists x \in \gamma(\phi_1) \colon \gamma(\text{slice}(\phi_2, S))$$

  It follows that $\gamma(\phi_1)$ is contains some x_0, for which M is a model of the formula $\gamma(\text{slice}(\phi_2, S))[x_0/x]$, or equivalently, $M' = (\mathcal{I}, \xi[x \mapsto x_0], s, s')$ is a model for $\gamma(\text{slice}(\phi_2, S))$.

  By the induction hypothesis there exists a $Br \in \text{Branches}(\phi_2)$, for which M' is a model of $\gamma(\text{slice}(\phi_2, Br))$. Equivalently, M is a model of $\gamma(\text{slice}(\phi_2, Br))[x_0/x]$. This, in turn, implies that M is also a model of $\exists x \in \gamma(\phi_1) \colon \gamma(\text{slice}(\phi_2, Br))$, since x_0 was chosen from $\gamma(\phi_1)$. Therefore it follows that M is a model for $\gamma(\text{slice}(\phi, Br))$. Note that Br is a branch of ϕ by Lemma 5.

  Assume additionally that $Br_1, \ldots, Br_k \in [B]_A$ for some assignment strategy A and some equivalence class $[B]_A$ of $\sim_A$. As all branches $Br_1, \ldots, Br_k$ are branches of ϕ_2, the induction hypothesis guarantees that Br is equivalent to Br_1 and, by transitivity, to all of the branches $Br_1, \ldots, Br_k$.

- $\phi = \ell :: \text{IF } \phi_1 \text{ THEN } \phi_2 \text{ ELSE } \phi_3 :$

  Assume, as the induction hypothesis, that the proposition holds for ϕ_2 and ϕ_3. By definition,

  $$\text{slice}(\phi, S) = \text{IF } \phi_1 \text{ THEN } \text{slice}(\phi_2, S) \text{ ELSE } \text{slice}(\phi_3, S)$$

  Assume, that M is a model of $\gamma(\text{slice}(\phi, S))$. Then,

  $$\gamma(\text{slice}(\phi, S)) = \text{IF } \gamma(\phi_1) \text{ THEN } \gamma(\text{slice}(\phi_2, S)) \text{ ELSE } \gamma(\text{slice}(\phi_3, S))$$

  By Lemma 10, we know that

  $$\gamma(\text{slice}(\phi, S)) = \text{IF } \gamma(\phi_1) \text{ THEN } \gamma(\text{slice}(\phi_2, S \cap \text{Labs}(\phi_2))) \text{ ELSE } \gamma(\text{slice}(\phi_3, S \cap \text{Labs}(\phi_3)))$$

By Lemma 6, branches of ϕ are either branches of ϕ_2 or of ϕ_3. We have two options, either $\gamma(\phi_1)$ is true under M, or it isn't. If $\gamma(\phi_1)$ is true under M, then, as M is a model of $\gamma(\text{slice}(\phi, S))$, we can conclude that M is a model of $\gamma(\text{slice}(\phi_2, S \cap \text{Labs}(\phi_2)))$ and $S \cap \text{Labs}(\phi_2)$ is not empty. It is therefore a union of $0 < l \leq k$ branches $Br'_1, \ldots, Br'_l$ from Branches(ϕ_2). By the induction hypothesis for ϕ_2, we know that there exists a $Br^2 \in$ Branches(ϕ_2), for which M is a model of $\gamma(\text{slice}(\phi_2, Br^2))$. By Lemma 6, Br^2 is also a branch for ϕ.

Assume additionally that $Br_1, \ldots, Br_k \in [B]_A$ for some assignment strategy A and some equivalence class $[B]_A$ of $\sim_A$. Trivially, all branches $Br'_1, \ldots, Br'_l$ are equivalent as well, as they each equal some branch Br_j. Therefore, $Br^2 \sim_A Br'_1$. As Br'_1 is equal to one of the branches $Br_1, \ldots, Br_k$, which are all equivalent, we see that Br^2 is equivalent to Br_1 and, by transitivity, to all of the branches $Br_1, \ldots, Br_k$.

The case where $\gamma(\phi_1)$ is false under M is proven analogously.

We conclude, that the proposition holds for all ϕ. □

Corollary 1. *Let ϕ be an α-TLA$^+$ expression and A an assignment strategy for ϕ. For every equivalence class $[B]_A$ of $\sim_A$, the following holds: Using the set $X = \bigcup_{Y \in [B]_A} Y$, if there exists a model M of $\gamma(\text{slice}(\phi, X))$, then M must be a model of $\gamma(\text{slice}(\phi, Br))$, for some branch $Br \in [B]_A$.*

Theorem 2 allows us to use symbolic transitions, not individual branches:

Theorem 2. *Let ϕ be an α-TLA$^+$ expression and A an assignment strategy for ϕ. There is a model M of the TLA$^+$ formula $\gamma(\phi)$ if and only if there exists a $Br \in$ Branches(ϕ), such that M is a model of $\gamma(\psi)$, where ψ is the symbolic transition generated by Br and A.*

Proof. First, assume that there exists a model M of $\gamma(\phi)$. By Proposition 6, we know that there exists a branch Br, for which M is a model of $\gamma(\text{slice}(\phi, Br))$. Then, denote by Y the set $\bigcup_{Z \in [Br]_A} Z$. Obviously, $Br \in [Br]_A$, so $Br \cup Y = Y$. It follows, by Proposition 8, that M is a model of $\gamma(\text{slice}(\phi, Y))$. However, slice($\phi$, Y) is the symbolic transition generated by Br and A, by definition, so the implication holds.

Next, assume that there exists a model M of $\gamma(\psi)$, for a symbolic transition ψ. There exists an equivalence class $[B]_A$, such that ψ has the shape slice(ϕ, Y) where Y is taken to be $\bigcup_{Br \in [B]_A} Br$. By Corollary 1 of Proposition 9, we know that there exists a branch $Br \in [B]_A$, for which M is a model of $\gamma(\text{slice}(\phi, Br))$. □

6.7 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

MODULE *max*

EXTENDS *Naturals*
VARIABLE tok, max, id
$Init \triangleq tok = 1 \land id \in [1..3 \to Nat] \land max = 0$
$P(i) \triangleq tok = i \land tok' = 1 + i \,\%\, 3 \land max' = \text{IF } id[i] > max \text{ THEN } id[i] \text{ ELSE } max$
$Next \triangleq (P(1) \lor P(2) \lor P(3)) \land id' = id$

Figure 6.3: A distributed maximum computation in a ring of three processes in TLA$^+$

symbolic transitions from TLA$^+$ specifications, or report that none exist. It uses Z3 as the background SMT solver.

We have chosen specifications both from publicly available sources, for example EWD840 and Paxos, and from a collection of algorithms we have encoded in TLA$^+$ ourselves. All of these specifications are available in [tlaa]. For each specification, we focus on the *Next* formula. We report the number of subexpressions in $\alpha(Next)$, that is, $|\,\mathrm{Sub}(\alpha(Next))|$, 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 6.7. Note that though we omit the specification in Figure 6.1 from the experiments, due to its simplicity, the outcome for it is 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 6.1, 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 6.3. 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 ith 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 6.4 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 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 the efficiency of bounded model

Table 6.7: Experimental results. The meaning of columns is as follows: "SE" is the number of subexpressions, "ST" is the size of the assignment strategy, "TR" is the number of symbolic transitions, and "time" is the time in milliseconds.

Specification	**SE**	**ST**	**TR**	**time**
Aba_asyn_byz	184	48	8	91
AlternatingBit	183	49	7	169
Bakery	303	71	16	260
BcastByz	96	22	5	53
BcastFolklore	75	17	4	14
Bosco	114	18	9	29
Boulanger	353	85	18	277
C1cs	171	37	8	76
Cbc_max	265	72	9	142
Cf1s_folklore	258	69	14	139
ChangRoberts	88	18	7	56
Channel	14	2	2	142
DieHard	43	12	6	28
DijkstraMutex	305	75	18	271
EWD840	79	16	4	47
HourClock	3	1	1	19
LamportMutex	130	30	6	75
MissionariesAndCannibals	20	2	1	44
Nbacc_ray97	77	15	14	48
Nbacg_guer01	296	82	13	220
Paxos	92	16	4	49
PaxosCommit	153	28	10	179
Prisoners	60	14	4	34
Queens	19	4	2	34
Raft	841	222	23	932
SchedulingAllocator	73	12	4	19
SimpleAllocator	40	6	3	35
Spanning	74	12	4	27
TCommit	24	3	3	22
TwoPhase	122	28	7	65
Voting	39	4	2	25

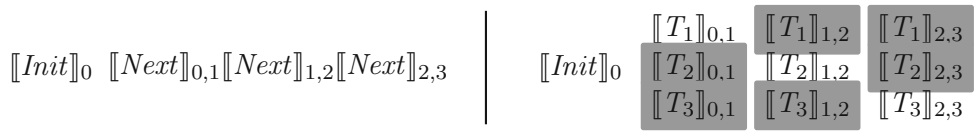


Figure 6.4: SMT formulas that are constructed when checking the executions up to length 4: using the action *Next* (left), and using symbolic transitions (right). The gray formulas are excluded from the SMT context during the exploration.

checking for C programs [BHvM09][Ch. 16], hence, we expect it to be effective for the verification of TLA$^+$ specifications too.

We implement the techniques presented in this chapter as a preprocessing step in our APLACHE [pro] model checker.

6.8 Conclusions

We have introduced a technique to compute symbolic transitions of a TLA$^+$ specification by finding expressions that can be interpreted as assignments. Importantly, we designed the technique with soundness in mind. Our results can be used as a first preprocessing step for a symbolic model checker or a type checker 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 simple errors. Hence, most of the benchmarks already come in a form compatible with TLC. Thus, we expect our approach to also work in practice.

Interestingly, our solution can be applied in the more specific context of Boolean circuits and Binary Decision Diagrams (BDDs). The model checker NuSMV [CCG$^+$02] supports non-deterministic assignments with similar requirements on the assignment orders. On one hand, NuSMV provides the user with explicit assignment syntax, and therefore requires only a check to see, whether the assignments satisfy the correct properties, instead of finding the assignments altogether. On the other hand, the user may specify the transition relation as a formula over new and old variables. Our technique may help in finding assignments in such a formula, if it is possible. Finally, Cabodi et. al [CCLQ97] use disjunctive partitioning to reduce BDD size and allow for the verification of large circuits. Our symbolic transitions can be seen as a disjunctive partitioning of a TLA$^+$ formula.

CHAPTER 7

TLA$^+$ model checking made symbolic

This chapter is an extension of the work presented in [KKT19b]. The techniques described in it were implemented by the author and advisor in the Apalache model checker.

7.1 Introduction

As TLA was initially designed for writing mathematical proofs about algorithms, it did not offer a concrete syntax for their specification. Rather, the algorithm designers were expected to present their algorithms in first-order logic and choose a convenient interpretation.

While there has been progress made towards proof automation in TLAPS 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 [NRZ+15, Ong14].

In the industry, [NRZ+15] and [Gus19] reported on finding real bugs by checking TLA$^+$ specifications by running the TLC model checker.

Our Approach. We are developing a more efficient 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.

3. Following our previous work [KTK18], we assume that for each variable x, there is a set of expressions $x' = e$ and $x' \in S$ that can be 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 work 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 powersets, 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 KerA$^+$ to capture all but few TLA$^+$ operators over finite structures.

2. We define operational semantics of KerA$^+$ in terms of reduction rules. Given a KerA$^+$ formula ϕ, the reduction system produces SMT constraints that are equisatisfiable to ϕ.

3. We prove soundness of the rewriting rules.

4. We show how to use the reduction system for: (a) checking inductive invariants, and (b) checking safety of TLA$^+$ specifications by bounded model checking.

5. We implement the rewriting system and run experiments on a number of benchmarks from the public TLA$^+$ repository. The experiments demonstrate that our tool APALACHE is often more efficient than TLC on the benchmarks with large state spaces.

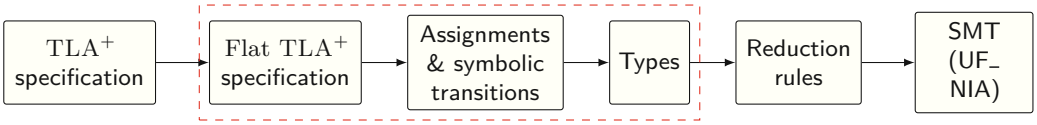


Figure 7.1: The basic workflow of APALACHE. The dashed area shows the preprocessing phases.

Figure 7.1 shows the main phases of APALACHE [KKT19a]. First, the call sites of user-defined operators are replaced with their bodies, which produces a flat specification. Second, the technique by [KTK18] 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 of the chapter. We begin with a motivating example in Section 7.2. We discuss the preprocessing steps in Section 7.3. In Section 7.4, we introduce the kernel language KERA$^+$. We introduce the reduction framework in Section 7.5 and the reduction rules in Sections 7.6–7.10. The soundness of the framework is discussed in Section 7.11. In Sections 7.12 and 7.13, we discuss the implementation and the experiments. We finish with the discussion of related and future work in Sections 7.14 and 7.15.

7.2 Example: the two-phase commit protocol in TLA+

A comprehensive manual on TLA$^+$ can be found in the book by [Lam02]. We introduce typical TLA$^+$ constructs by discussing the famous two-phase commit protocol by [LS79]. 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 7.2 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.

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 RMs from which the TM has received "Prepared" messages
$msgs$	The set of all messages sent in the distributed system

$Init \triangleq \land rmState = [rm \in RM \mapsto \text{"working"}]$ constraints on the initial states
$\land tmState = \text{"init"} \land tmPrepared = \{\} \land msgs = \{\}$

The transitions by the transaction manager and the resource managers:

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

The transaction manager commits the transaction:

$TMCommit \triangleq tmState = \text{"init"} \land tmPrepared = RM \land tmState' = \text{"committed"}$
$\land msgs' = msgs \cup \{[type \mapsto \text{"Commit"}]\} \land \text{UNCHANGED } \langle rmState, tmPrepared\rangle$

The transaction manager spontaneously aborts the transaction:

$TMAbort \triangleq tmState = \text{"init"} \land tmState' = \text{"aborted"}$
$\land msgs' = msgs \cup \{[type \mapsto \text{"Abort"}]\} \land \text{UNCHANGED } \langle rmState, tmPrepared\rangle$

Resource manager rm prepares:

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

Resource manager rm spontaneously decides to abort:

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

Resource manager rm is told by the TM to commit:

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

Resource manager rm is told by the TM to abort:

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

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
$TMRcvPrepared(rm) \lor RMPrepare(rm) \lor RMRcvCommitMsg(rm)$
$\lor RMChooseToAbort(rm) \lor RMRcvAbortMsg(rm)$

Figure 7.2: The two-phase commit protocol in TLA$^+$ as specified in [GL06] (We have only changed the indentation and comments to save some space).

- The variable *msgs* stores the set of messages sent by the managers. It contains records of three kinds: $[type \mapsto \text{“Commit”}]$, $[type \mapsto \text{“Abort”}]$, and $[type \mapsto \text{“Prepared”}, rm \mapsto S]$. 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 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 capture the state before a transition, while the latter capture 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 $[type \mapsto \text{“Prepared”}, \text{rm} \mapsto rm]$ to the set of messages *msgs*. Finally, the action requires that $tmState' = tmState$ and $tmPrepared' = tmPrepared$, as indicated by UNCHANGED $\langle tmState, tmPrepared \rangle$.

The algorithm is designed to satisfy the following invariant:

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

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, for example, 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.

In Section 7.3, we discuss how to deal with these issues before doing the translation to SMT.

7.3 Preprocessing: flattening, assignments, and types

7.3.1 Flattening

As exemplified by Section 7.2, TLA$^+$ specifications are normally written as a collection of operator definitions. They can be also organized in modules. As the operator *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 LET-IN expression. We also instantiate modules, in order to obtain a single-module specification, in which the operators *Init* and *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 version 2 of TLA$^+$. Hence, the users can conveniently write expressions in terms of recursion 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.

7.3.2 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.

Recently, we introduced a symbolic technique for finding such assignments without evaluating the TLA$^+$ formula [KTK18]. 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.

Example 4. *Consider the example in Figure 7.2. 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.*

7.3.3 Types

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

$$\tau ::= \mathsf{Name} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \tau \to \tau \mid \mathsf{Set}(\tau) \mid \mathsf{Seq}(\tau) \mid \tau * \cdots * \tau \mid [nm_1 : \tau, \ldots, nm_k : \tau]$$

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.

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 an appropriate type, based on its shape. For instance, the literals $0, 1, -1, \ldots$ have type Int, and the literals FALSE and TRUE have type 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., $+ : \mathsf{Int} * \mathsf{Int} \to \mathsf{Int}$. 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 7.2, records $[type \mapsto \text{“}\mathsf{Abort}\text{”}]$ and $[type \mapsto \text{“}\mathsf{Prepared}\text{”}, rm \mapsto rm]$ are both added to the set $msgs$. The user has to annotate the records and their sets with a super type, e.g., $[type : \mathsf{Name}, rm : \mathsf{Set}(\mathsf{Name})]$.

7.4 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 UNION $\{S_1, \ldots, S_n\}$, which constructs

Table 7.1: The language KerA$^+$. We highlight the expressions that do not have counterparts in pure TLA$^+$.

Literals:	FALSE, TRUE	0,1,-1,2,-2,...	$c_1, \ldots, c_n$ (*constants*)
Integers:	$i_1 \bullet i_2$ where $\bullet$ is one of: $+, -, *, \div, \%, <, \leq, >, \geq, =, \neq$		
Sets:	$\{e_1, \ldots, e_n\}$	$\{x \in S : p\}$	$\{e : x \in S\}$
	UNION S	$i_1 \,..\, i_2$	$Cardinality(S)$
	$x \in [S_1 \to S_2]$	$x \in$ SUBSET S	
Control:	ITE(p, e_1, e_2)	$e_1 \oplus \ldots \oplus e_n$	$x' \in S$
	$x' \in [S_1 \to S_2]$	$x' \in$ SUBSET S	
Quantifiers:	$\exists x \in S : p$	CHOOSE $x \in S : p$	FROM $e_1, \ldots, e_n$ BY θ
Functions:	$[x \in S \mapsto e]$	$f[e]$	DOMAIN f
	$[f$ EXCEPT $![e_1] = e_2]$		
Records:	$[nm_1 \mapsto e_1, \ldots, nm_n \mapsto e_n]$		DOMAIN r
	$e.nm$		
Tuples:	$\langle e_1, \ldots, e_n \rangle$	$t[i]$	DOMAIN t
Sequences:	$\langle e_1, \ldots, e_n \rangle$	$s[i]$	DOMAIN s
	$[s$ EXCEPT $![i] = e]$	$Len(s)$	$s \circ t$
	$Head(s)$, $Tail(s)$	$SubSeq(s, i, j)$	

the union $S_1 \cup \cdots \cup S_n$. The binary operator $S_1 \cup S_2$ is equivalent to UNION $\{S_1, S_2\}$. We add a few auxiliary operators that simplify the translation.

A list of KerA$^+$ expressions is given in Table 7.1. It might seem surprising that very basic operators such as Boolean operators are missing. In fact, they can be expressed with IF-THEN-ELSE:

$$\neg p \equiv \text{ITE}(p, \text{FALSE}, \text{TRUE}) \qquad p \wedge q \equiv \text{ITE}(p, q, \text{FALSE}) \qquad p \vee q \equiv \text{ITE}(p, \text{TRUE}, q)$$

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

- *Assignment* $x' \in S$: Following TLC, under the conditions given by [KTK18], 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 $x' \in S$.
- *Non-deterministic disjunction* $\phi_1 \oplus \ldots \oplus \phi_n$: This operator formalizes the special form of TLC disjunction. It evaluates to true if and only if the disjunction $\phi_1 \vee \cdots \vee \phi_n$ evaluates to true. However, non-deterministic disjunction adds constraints on the variable assignments: For every $i, j \in 1..n$ and $i \neq j$, formula ϕ_i contains an assignment to a variable x' if and only if formula ϕ_j contains an assignment to x'. Note that this property is implied by the single-assignment property of symbolic transitions (see Section 7.3.2). Hence, we use it to compose the symbolic transitions.

- *Choice with an oracle* FROM $e_1, \ldots, e_n$ BY θ: This operator returns expression e_i when $\theta = i$ and $1 \leq i \leq n$; otherwise, it returns an arbitrary value of the same type as $e_1, \ldots, e_n$.

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 [Mer08b]. In Sections 7.6–7.10, we give a brief description of each KERA$^+$ operator along with the semantics for finite structures in terms of rewriting rules.

7.5 Rewriting framework

Our goal is to translate a KERA$^+$ expression into an equisatisfiable quantifier-free SMT formula. To this end, we introduce an abstract reduction system that allows us to iteratively transform a KERA$^+$ 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 KERA$^+$ 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 KERA$^+$ expression into a so-called cell that captures the result of symbolically evaluating the expression. The rewriting process terminates, when the input KERA$^+$ formula ϕ 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 ϕ.

7.5.1 Cells

In our framework, a cell is simply a first-order constant that is annotated with a type τ: 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_{name} to refer to a cell. We assume fixed a finite set of cells $\mathcal{C}$, which contains sufficiently many elements for rewriting a KERA$^+$ expression.

New cells are introduced when rewriting a KERA$^+$ expression. For example, the expression $\{1, 2\}$ is rewritten by a series of rewriting steps: $\{1, 2\} \rightsquigarrow \{\mathsf{c}_1, 2\} \rightsquigarrow \{\mathsf{c}_1, \mathsf{c}_2\} \rightsquigarrow \mathsf{c}_3$. We give the precise definition of $\rightsquigarrow$ in Section 7.5.4. While the original expression does not contain cells, the rewritten expressions do. In fact, cells are well-formed KERA$^+$ expressions, as they can be seen as KERA$^+$ constants. Hence, the introduced cells can be seen as: (1) first-order constants in SMT, and (2) KERA$^+$ constants in KERA$^+$, which would be introduced in TLA$^+$ using the string notation, e.g., "`abc`".

7.5.2 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 $\mathsf{c}_1 \xrightarrow{i}_{\mathcal{A}} \mathsf{c}_2$, we denote that $(\mathsf{c}_1, i, \mathsf{c}_2) \in \mathcal{E}$. Similarly, we write $\mathsf{c} \to_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n$ to say that c points to $\mathsf{c}_1, \ldots, \mathsf{c}_n$ in this order, that is, $\mathsf{c} \xrightarrow{i}_{\mathcal{A}} \mathsf{c}_i$ for $1 \leq i \leq n$ and for every $\mathsf{c}' \in \mathcal{V}(\mathcal{A})$ and $j > n$, it holds that $(\mathsf{c}, j, \mathsf{c}') \notin \mathcal{E}(\mathcal{A})$. We use the following notation to extend an arena $\mathcal{A}$:

- Notation $\mathcal{A}, \mathsf{c} : \tau$ 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 τ, i.e., $\mathsf{c} \notin \mathcal{V}(\mathcal{A})$.
- Notation $\mathcal{A}, \mathsf{c} \to \mathsf{c}_1, \ldots, \mathsf{c}_n$ to introduce the arena $\mathcal{A}'$ such that $\mathcal{V}(\mathcal{A}') = \mathcal{V}(\mathcal{A})$ and $\mathcal{E}(\mathcal{A}') = \mathcal{E}(\mathcal{A}) \cup \{(\mathsf{c}, i, \mathsf{c}_i) \mid 1 \leq i \leq n\}$.

Example 5. *Figure 7.3 shows examples of memory arenas for several* KERA$^+$ *expressions. In example (a), the arena contains six cells: three cells of type* Int *that represent integers 1, 2, 3; two cells of type* Set(Int) *that represent the sets* $\{1, 2\}$ *and* $\{2, 3\}$*; and one cell of type* Set(Set(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 7.5.3.*

In example (b), the arena contains five cells: three cells to encode the integers and string, the cell c_{14} *to encode the record* $[b \mapsto 0, c \mapsto 3]$*, and the cell* c_{15} *to encode the tuple* $\langle$*"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* $\mathsf{c}_{15} \xrightarrow{1} \mathsf{c}_{11}$ *and the tuple type* $\mathsf{Name} * \mathsf{Int} * [b : \mathsf{Int}, c : \mathsf{Int}]$*, 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 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* $\{(x, f[x]) \colon x \in \text{DOMAIN } f\}$*. 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*

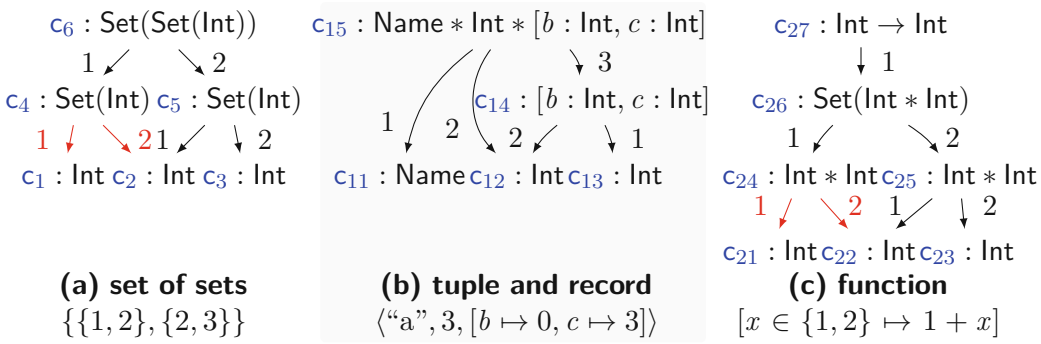


Figure 7.3: Examples of arenas for data structures in KERA$^+$. The leaf cells are equal to the following constants: $c_1 = c_{21} = 1$, $c_2 = c_{22} = 2$, $c_3 = c_{23} = 3$, $c_{11} = a$, $c_{12} = 3$, and $c_{13} = 0$

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.

7.5.3 SMT constraints

We recapitulate the necessary notions related to many-sorted first-order logic. We assume fixed a set of *sorts* $\mathcal{S}$, which includes exactly one sort s_τ per type τ that is defined in Section 7.3. Further, let $\mathcal{F}$ be a set of *functional symbols*, each functional symbol is assigned a non-negative arity. For convenience, we say that the set of cells $\mathcal{C}$ coincides with the set of functional symbols of arity 0 from the set $\mathcal{F}$. Each symbol $f \in \mathcal{F}$ is assigned a sort $sort(f) \in \mathcal{S}$. The *ground terms* are defined as follows: (1) every constant $c \in \mathcal{C}$ is a ground term, and (2) if $t_1, \ldots, t_n$ are ground terms and $f \in \mathcal{F}$ 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 $\mathcal{P} \subseteq \mathcal{F}$, which contains the symbols that are assigned a sort $s_{\tau_1 \times \cdots \times \tau_n \to \mathsf{Bool}}$ 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 $\mathcal{F}$ contains the standard symbols of integer arithmetic along with uninterpreted functions, and their interpretation is standard. In particular, the sorts s_{Bool} and s_{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.)

Encoding Arenas in SMT When rewriting a 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 $en\langle e\rangle$, whose value indicates, whether the edge e is enabled or not.

Example 6. *Consider the edge $e_{41} = (c_4, 1, c_1)$ in Figure 7.3 (a). If $en\langle e_{41}\rangle$ 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.*

7.5.4 Abstract reduction system (ARS)

We assume fixed a finite set of variables *Vars* that are used in 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 KERA$^+$ expression, possibly containing cells,
- $\mathcal{A}$ is an arena,
- ν is a partial function from *Vars* to $\mathcal{V}(\mathcal{A})$, which is called *binding*, and
- Φ is a set of first-order formulas, which represents SMT constraints.

We define $\rightsquigarrow$ via a set of reduction rules. For instance, the rules (BOOL) and (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 new state of the reduction system below the bar. By convention, the state is always written as the first premise, using the notation $\langle e \mid \mathcal{A} \mid \nu \mid \Phi\rangle$.

$$\frac{\langle b \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad b \text{ is FALSE or TRUE}}{\langle \mathsf{c} \mid \mathcal{A}, \mathsf{c} : \mathsf{Bool} \mid \nu \mid \Phi, \mathsf{c} = b\rangle}\ (\textsc{Bool})$$

$$\frac{\langle n \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad n \text{ is } 0, 1, -1, \ldots}{\langle \mathsf{c} \mid \mathcal{A}, \mathsf{c} : \mathsf{Int} \mid \nu \mid \Phi, \mathsf{c} = n\rangle}\ (\textsc{Int})$$

$$\frac{\langle \mathsf{c}_\ell \bowtie \mathsf{c}_r \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad \bowtie \text{ is one of } <, \leq, >, \geq, =, \neq}{\langle \mathsf{c}_{res} \mid \mathcal{A}, \mathsf{c}_{res} : \mathsf{Bool} \mid \nu \mid \Phi, \mathsf{c}_{res} \leftrightarrow \mathsf{c}_\ell \bowtie \mathsf{c}_r\rangle}\ (\textsc{IntCmp})$$

Once we have introduced integer cells for the literals, we can reduce integer comparisons using the rule (INTCMP) and reduce integer arithmetics using the rule (INTARITH). The reduction rules add new SMT constraints to the set Φ.

$$\frac{\langle \mathsf{c}_\ell \circ \mathsf{c}_r \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad \circ \text{ is one of } +, -, *, \div, \%}{\langle \mathsf{c}_{res} \mid \mathcal{A}, \mathsf{c}_{res} : \mathsf{Int} \mid \nu \mid \Phi, \mathsf{c}_{res} = \mathsf{c}_\ell \circ \mathsf{c}_r \rangle} \text{ (IntArith)}$$

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 $\{x \in S : p\}$.

$$\frac{\begin{array}{c}\langle Op(e_1, \ldots, e_n) \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0 \rangle \\ \langle e_i \mid \mathcal{A}_{i-1} \mid \nu_{i-1} \mid \Phi_{i-1} \rangle \\ \langle \mathsf{c}_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle \qquad \text{for } 1 \leq i \leq n\end{array}}{\langle Op(\mathsf{c}_1, \ldots, \mathsf{c}_n) \mid \mathcal{A}_n \mid \nu_n \mid \Phi_n \rangle} \text{ (RedArg)}$$

To apply the reduction system to a KerA$^+$ expression e, e.g., to *Init* and *Next*, we introduce an initial state $\langle e_0 \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0 \rangle$, whose arena, binding, and SMT constraints are empty. Formally, $\mathcal{A}_0 = (\emptyset, \emptyset)$, $\Phi_0 = \emptyset$, and $\nu(x) = \bot$ for $x \in \mathit{Vars}$. Usually, the expression e_0 is a formula, that is, it has type Bool. For simplicity, we also assume that all constants that appear in e_0 have basic types, that is, Int, Bool, and 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_{term} is reached, and the terminal cell c_{term} has type Bool, we add the assertion c_{term} to the SMT constraints and check their satisfiability. In Sections 7.6–7.10, we introduce rewriting rules for sets, functions, tuples, records, sequences, and control operators. Section 7.11 contains soundness proofs.

7.6 Sets

Sets lie in the theoretical foundation of TLA$^+$, as it builds upon Zermelo-Fränkel 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_{set}. The rule links the elements $\mathsf{c}_1, \ldots, \mathsf{c}_n$ to c_{set} in the arena and adds the constraint $en\langle \mathsf{c}_{set}, i, \mathsf{c}_i \rangle$ for each $1 \leq i \leq n$. Several important observations should be made. First, we only add constraints on the edges from c_{set} to the cells $\mathsf{c}_1, \ldots, \mathsf{c}_n$, 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.

$$\frac{\langle \{\mathsf{c}_1, \ldots, \mathsf{c}_n\} : \mathsf{Set}(\tau) \mid \mathcal{A} \mid \nu \mid \Phi \rangle}{\langle \mathsf{c}_{set} : \mathsf{Set}(\tau) \mid \mathcal{A}, \mathsf{c}_{set}, \mathsf{c}_{set} \to \mathsf{c}_1, \ldots, \mathsf{c}_n \mid \nu \mid \Phi, \bigwedge_{1 \leq i \leq n} en\langle \mathsf{c}_{set}, i, \mathsf{c}_i \rangle \rangle} \text{ (ENUM)}$$

Set Membership. An expression $\mathsf{c}_x \in \mathsf{c}_S$ such that $\mathsf{c}_S \to_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n$ is reduced to $\bigvee_{1 \leq i \leq n} \mathsf{c}_x = \mathsf{c}_i$.

Set Filter. An expression $\{x \in S \colon p\}$ constructs the set T that has only the elements of S that satisfy the predicate p.

$$\frac{\begin{array}{c}\langle \{x \in \mathsf{c}_S \colon p\} : \mathsf{Set}(\tau) \mid \mathcal{A} \mid \nu \mid \Phi \rangle \\ \langle p[\mathsf{c}_1/x], \ldots, p[\mathsf{c}_n/x] \mid \mathcal{A} \mid \Phi \mid \nu \rangle \\ \hline \langle \mathsf{c}_1^p, \ldots, \mathsf{c}_n^p \mid \mathcal{A}' \mid \Phi' \mid \nu' \rangle \qquad \mathsf{c}_S \to_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n\end{array}}{\langle \mathsf{c}_T : \mathsf{Set}(\tau) \mid \mathcal{A}', \mathsf{c}_T \to \mathsf{c}_1, \ldots, \mathsf{c}_n \mid \nu' \mid \Phi', \mathit{InFilter} \rangle} \text{ (FILTER)}$$

The rule (FILTER) implements this semantics in two steps. First, it reduces the applications of predicate p to all potential set elements $\mathsf{c}_1, \ldots, \mathsf{c}_n$, that is, it rewrites the expressions $p[\mathsf{c}_i/x]$ 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 (*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:

$$en\langle \mathsf{c}_T, i, \mathsf{c}_i \rangle \leftrightarrow (\mathsf{c}_i^p \wedge en\langle \mathsf{c}_S, i, \mathsf{c}_i \rangle) \text{ for } 1 \leq i \leq n \qquad (\mathit{InFilter})$$

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.

$$\frac{\begin{array}{c}\langle \text{UNION } \mathsf{c}_S \colon \mathsf{Set}(\mathsf{Set}(\tau)) \mid \mathcal{A} \mid \nu \mid \Phi \rangle \\ \mathsf{c}_S \to_{\mathcal{A}} \mathsf{c}_S^1, \ldots, \mathsf{c}_S^n \qquad \mathsf{c}_S^i \to_{\mathcal{A}} \mathsf{c}_1^i, \ldots \mathsf{c}_{m_i}^i \text{ for } 1 \leq i \leq n\end{array}}{\langle \mathsf{c}_U \colon \mathsf{Set}(\tau) \mid \mathcal{A}, \mathsf{c}_U, \mathsf{c}_U \to \mathsf{c}_1^1, \ldots, \mathsf{c}_{m_1}^1, \mathsf{c}_1^2, \ldots, \mathsf{c}_{m_2}^2, \ldots, \mathsf{c}_{m_n}^n \mid \nu \mid \Phi, \ \mathit{InU} \rangle} \text{ (UNION)}$$

The SMT constraint (*InU*) simply requires a cell c_j^i to be in c_U if and only of it is in the set containing it, that is, in c_S^i, and the set c_S^i belongs to c_U:

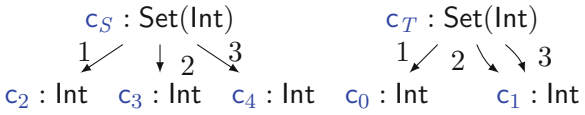


Figure 7.4: An arena constructed for the set comprehension $\{x \div 3 : x \in \{2,3,4\}\}$. Every cell c_i has value i for $0 \leq i \leq 4$. Cell c_S encodes the set $\{2,3,4\}$, and cell c_T encodes the result of the set comprehension.

$$en\langle \mathsf{c}_U, idx_{i,j}, \mathsf{c}_j^i \rangle \leftrightarrow \Big(en\langle \mathsf{c}_S^i, j, \mathsf{c}_j^i \rangle \wedge en\langle \mathsf{c}_S, i, \mathsf{c}_S^i \rangle \Big) \text{ for } 1 \leq i \leq n, 1 \leq j \leq m_i, \qquad (InU)$$

where the edge index $idx_{i,j}$ is defined as $m_1 + \cdots + m_{i-1} + j$.

The constraint (InU) may seem to be unsound. Indeed, consider the arena in Figure 7.3 (a) and assume that we compute UNION c_6. Further, assume that the SMT solver sets $en\langle \mathsf{c}_5, 1, \mathsf{c}_2 \rangle$ to true and $en\langle \mathsf{c}_4, 2, \mathsf{c}_2 \rangle$ 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 (InU) produces the following constraints (among others): $en\langle \mathsf{c}_U, 2, \mathsf{c}_2 \rangle \leftrightarrow en\langle \mathsf{c}_4, 2, \mathsf{c}_2 \rangle \wedge en\langle \mathsf{c}_6, 1, \mathsf{c}_4 \rangle$ and $en\langle \mathsf{c}_U, 3, \mathsf{c}_2 \rangle \leftrightarrow en\langle \mathsf{c}_5, 1, \mathsf{c}_2 \rangle \wedge en\langle \mathsf{c}_6, 2, \mathsf{c}_5 \rangle$. As a result, $en\langle \mathsf{c}_U, 2, \mathsf{c}_2 \rangle$ is false, whereas $en\langle \mathsf{c}_U, 3, \mathsf{c}_2 \rangle$ 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, $(\mathsf{c}_U, 3, \mathsf{c}_2)$.

Set Map. By definition, $\{e : x \in S\}$ constructs the set T with the following property: For every z, 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 7.4 shows the arena that is constructed in the process of reduction.

$$\frac{\langle \{e : x \in \mathsf{c}_S\} \mid \mathcal{A} \mid \nu \mid \Phi \rangle \text{ and } \mathsf{c}_S \rightarrow_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n \qquad \dfrac{\langle e[\mathsf{c}_1/x], \ldots, e[\mathsf{c}_n/x] \mid \mathcal{A} \mid \Phi \mid \nu \rangle}{\langle \mathsf{c}_1^e : \tau, \ldots, \mathsf{c}_n^e : \tau \mid \mathcal{A}' \mid \Phi' \mid \nu' \rangle}}{\langle \mathsf{c}_T \mid \mathcal{A}', \mathsf{c}_T : \mathsf{Set}(\tau), \mathsf{c}_T \rightarrow \mathsf{c}_1^e, \ldots, \mathsf{c}_n^e \mid \nu' \mid \Phi', InMap \rangle} \text{ (MAP)}$$

The rule works in two steps. First, it reduces the applications of expression e to all potential set elements $\mathsf{c}_1, \ldots, \mathsf{c}_n$, that is, it rewrites the expressions $e[\mathsf{c}_i/x]$ to c_i^e for $1 \leq i \leq n$. Second, the constraint $(InMap)$ enforces that a cell c_i^e 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:

$$en\langle \mathsf{c}_T, i, \mathsf{c}_i^e \rangle \leftrightarrow en\langle \mathsf{c}_S, i, \mathsf{c}_i \rangle \text{ for } 1 \leq i \leq n \qquad (InMap)$$

Example 7. *Consider Figure 7.4. 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 $en\langle \mathsf{c}_S, 3, \mathsf{c}_4 \rangle$*

to true and $en\langle \mathsf{c}_S, 2, \mathsf{c}_3\rangle$ *to false. Hence,* $en\langle \mathsf{c}_T, 3, \mathsf{c}_1\rangle$ *holds true and* $en\langle \mathsf{c}_T, 2, \mathsf{c}_1\rangle$ *does not. Still,* c_1 *belongs to the set encoded by* c_T*, as the edge* $(\mathsf{c}_T, 3, \mathsf{c}_1)$ *is enabled.*

Integer Interval $a..b$**.** This operator is quite often used in TLA$^+$ to define the set $\{i \in \mathbb{Z} \colon a \leq i \leq b\}$. The latter set cannot be defined in KERA$^+$, 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 $S \supseteq a..b$ that can be filtered by the KERA$^+$ expression $\{i \in S \colon a \leq i \wedge i \leq b\}$. 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 (SETEQ). Whenever two cells c_S and c_T are compared for the first time, (SETEQ) rewrites the definition of set equality into a Boolean cell c_{eq}. Additionally, it adds the SMT constraint $\mathsf{c}_S = \mathsf{c}_T \leftrightarrow \mathsf{c}_{eq}$, which allows us to use SMT equality in the later occurrences of $\mathsf{c}_S = \mathsf{c}_T$.

$$\frac{\begin{array}{c}\langle \mathsf{c}_S = \mathsf{c}_T \mid \mathcal{A} \mid \nu \mid \Phi\rangle \\ \langle (\forall x \in \mathsf{c}_S : x \in \mathsf{c}_T) \wedge (\forall x \in \mathsf{c}_T : x \in \mathsf{c}_S) \mid \mathcal{A} \mid \nu \mid \Phi\rangle\end{array}}{\begin{array}{c}\langle \mathsf{c}_{eq} : \mathsf{Bool} \mid \mathcal{A}' \mid \nu' \mid \Phi'\rangle \\ \hline \langle \mathsf{c}_{eq} \mid \mathcal{A}' \mid \nu' \mid \Phi', \mathsf{c}_S = \mathsf{c}_T \leftrightarrow \mathsf{c}_{eq}\rangle\end{array}}\ (\textsc{SetEq})$$

Set Cardinality. In TLA$^+$, an expression $\mathit{Cardinality}(S)$ 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, $\mathit{Cardinality}(S) \geq \mathit{Cardinality}(T)/2+1$, or to construct a set of integers $1..\mathit{Cardinality}(S)$, or as a function argument. Hence, we use a generic approach to computing the set cardinality by the recurrence relation in Equation (7.1), assuming that a set cell c_S is pointing to the element cells $\mathsf{c}_1, \ldots, \mathsf{c}_n$:

$$k_0 = 0 \quad \text{and} \quad k_{i+1} = \textsc{ite}(en\langle \mathsf{c}_S, i, \mathsf{c}_i\rangle \wedge \mathit{notSeen}_i, 1 + k_i, k_i) \text{ for } 0 < i \leq n \tag{7.1}$$

Equation (7.2) requires that the ith element contributes to the cardinality, if the previously considered elements are either outside of the set, or are different from the ith element:

$$\mathit{notSeen}_i = \bigwedge\nolimits_{1 \leq j < i} (en\langle \mathsf{c}_S, j, \mathsf{c}_j\rangle \rightarrow \mathsf{c}_j \neq \mathsf{c}_i) \text{ for } 0 < i \leq n \tag{7.2}$$

Hence, $\mathit{Cardinality}(\mathsf{c}_S) = k_n$. 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.

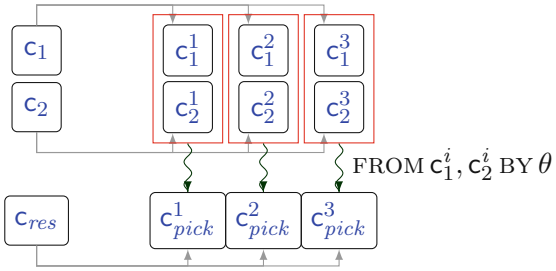


Figure 7.5: Picking a set among two set cells c_1 and c_2, pointing to c_1^1, c_1^2, c_1^3 and c_2^1, c_2^2, c_2^3 respectively. The result c_{pick} points to c_{pick}^1, c_{pick}^2, c_{pick}^3, which are picked from three sequences of two cells (in red boxes).

7.7 Picking set elements

While developing rewriting rules for TLA$^+$ operators, we found that many rules can be reduced to the auxiliary operator FROM $e_1, \ldots, e_n$ BY θ, where θ is an integer constant and $e_1, \ldots, e_n$ are TLA$^+$ expressions of the same type τ. The meaning of this operator is as follows: If $\theta \in 1..n$, then FROM $e_1, \ldots, e_n$ BY θ returns e_θ; Otherwise, it returns an arbitrary value of type τ. The constant θ 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 θ is not part of standard TLA$^+$. The syntax for TLA$^+$ proofs [Lam18] has a similar operator PICK $x \in S$, which returns an arbitrary element of the set S. However, 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 θ, 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_{pick} and requires that c_{pick} equals to the θth value as prescribed by the oracle. When the oracle has a value outside of $1..n$, the picked value is unconstrained.

$$\frac{\langle \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad c_1 : \tau, \ldots, c_n : \tau \qquad \tau \text{ is basic}}{\langle c_{pick} \mid \mathcal{A}, c_{pick} : \tau \mid \nu \mid \Phi, \bigwedge_{1 \le i \le n} (\theta = i \to c_{pick} = c_i) \rangle} \text{ (FROMBASIC)}$$

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 $\mathsf{Set}(\tau)$ for some type τ and $1 \le i \le n$. Without loss of generality, we assume that every cell points to exactly the same number of cells, that is, if $c_i \to_{\mathcal{A}} c_i^1, \ldots, c_i^k$ and $c_j \to_{\mathcal{A}} c_j^1, \ldots, c_j^m$, 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 $\mathsf{c}_i \to_{\mathcal{A}} \mathsf{c}_i^1, \ldots, \mathsf{c}_i^k, \ldots, \mathsf{c}_i^k$, where c_i^k 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_{pick}^j among the jth elements of the sets $\mathsf{c}_1, \ldots, \mathsf{c}_n$. Importantly, the operators FROM $c_1^j, \ldots, c_n^j$ BY θ are using the same oracle θ for every $j \in 1..m$. As a result, they pick the respective elements from the same set c_θ. Second, the resulting set c_{res} points to the picked elements $\mathsf{c}_{pick}^1, \ldots, \mathsf{c}_{pick}^m$.

$$\frac{\begin{array}{c}\langle \text{FROM } \mathsf{c}_1, \ldots, \mathsf{c}_n \text{ BY } \theta \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0 \rangle \\ \mathsf{c}_i : \mathsf{Set}(\tau) \text{ for } 1 \le i \le n \text{ and } \theta : \mathsf{Int} \qquad \mathsf{c}_i \to_{\mathcal{A}_0} \mathsf{c}_i^1, \ldots, \mathsf{c}_i^m \text{ for } 1 \le i \le n \\ \dfrac{\langle \text{FROM } \mathsf{c}_1^j, \ldots, \mathsf{c}_n^j \text{ BY } \theta \mid \mathcal{A}_{j-1} \mid \nu_{j-1} \mid \Phi_{j-1} \rangle}{\langle \mathsf{c}_{pick}^j : \tau \mid \mathcal{A}_j \mid \Phi_j \mid \nu_j \rangle \text{ for } 1 \le j \le m}\end{array}}{\langle \mathsf{c}_{res} \mid \mathcal{A}, \mathsf{c}_{res} : \mathsf{Set}(\tau), \mathsf{c}_{res} \to \mathsf{c}_{pick}^1, \ldots, \mathsf{c}_{pick}^m \mid \nu_m \mid \Phi_m, \mathit{InPicked} \rangle} \text{ (FROMSET)}$$

The constraint (InPicked) requires the new set cell c_{res} to contain a cell c_{pick}^j if and only if the respective set chosen by the oracle θ contains the jth cell.

$$en\langle \mathsf{c}_{res}, j, \mathsf{c}_{pick}^j \rangle \leftrightarrow \bigvee_{1 \le i \le n} \theta = i \wedge en\langle \mathsf{c}_i, j, \mathsf{c}_i^j \rangle \text{ for } 1 \le j \le m \qquad \text{(InPicked)}$$

Example 8. *Figure 7.5 shows an example of the rule applied to* FROM $\mathsf{c}_1, \mathsf{c}_2$ BY θ. *The cells* c_1 *and* c_2 *have type* $\mathsf{Set}(\tau)$, *each of them pointing to three element cells* c_i^1, c_i^2, *and* c_i^3 *for* $i \in \{1, 2\}$. *The rule first applies* FROM $\mathsf{c}_1^j, \mathsf{c}_2^j$ BY θ *three times for* $j \in \{1, 2, 3\}$ *to pick one element* c_{pick}^j *from each pair. Note that use of* θ *guarantees us that the elements are drawn from the same set. The resulting cell* c_{res} *is pointing to the three picked cells* c_{pick}^1, c_{pick}^2, *and* c_{pick}^3.

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 SUBSET S). They are similar to (FROMBASIC) and (FROMSET) and are omitted for brevity.

7.8 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.

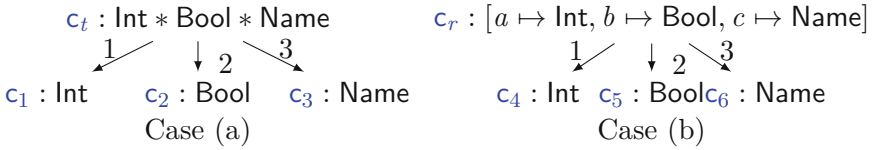


Figure 7.6: (a) The arena constructed for the tuple $\langle 1, \text{TRUE}, "abc" \rangle$, assuming that the expressions 1, TRUE, and "abc" were rewritten into cells c_1, c_2, and c_3. (b) The arena constructed for the record $[a \mapsto 1, b \mapsto \text{TRUE}, c \mapsto "abc"]$, assuming that the expressions 1, TRUE, and "abc" were rewritten into cells c_4, c_5, and c_6.

Tuple Constructor. A tuple constructor adds a new cell pointing to the element cells in their index order. Figure 7.6 (a) shows an example of applying the rule (TUPCTOR).

$$\frac{\langle \langle c_1, \ldots, c_n \rangle : \tau_1 * \cdots * \tau_n \mid \mathcal{A} \mid \nu \mid \Phi \rangle}{\langle c_{new} \mid \mathcal{A}, c_{new} : \tau_1 * \cdots * \tau_n, c_{new} \to c_1, \ldots, c_n \mid \nu \mid \Phi \rangle} \text{ (TUPCTOR)}$$

Tuple Application. The tuple application rule returns the ith cell pointed by the tuple cell:

$$\frac{\langle c_t[i] \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad c_t \to_{\mathcal{A}} c_1, \ldots, c_n \qquad i \in \{1, \ldots, n\}}{\langle c_i \mid \mathcal{A} \mid \nu \mid \Phi \rangle} \text{ (TUPAPP)}$$

Tuple Domain. For a tuple t of type $\tau_1 * \cdots * \tau_n$, the expression 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 DOMAIN r, which returns the set $\{nm_1, \ldots, nm_n\}$. Figure 7.6 (b) shows an example of rewriting a record constructor.

7.9 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 $\tau_1 \to \tau_2$, 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 $\{\langle x, f[x] \rangle : x \in \text{DOMAIN } f\}$. As a result, we reuse the rules for sets (Section 7.6) and tuples (Section 7.8). 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 7.3 (c) for example. We use the notation $\mathsf{funrel}(\mathsf{c}_f)$ 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 $\{\langle x, e\rangle : x \in S\}$. This rule produces a cell c_{rel} that encodes the associated relation c_{rel} of type $\mathsf{Set}(\tau_1 * \tau_2)$, where τ_1 is the type of elements of S, and τ_2 is the type of e. We add a cell c_f of type $\tau_1 \to \tau_2$ and make it point to c_{rel}, that is, $\mathsf{c}_f \to_{\mathcal{A}} \mathsf{c}_{rel}$. The rule (FUNCTOR) produces c_f as a result.

Function Domain (FunDom). Assuming that f is reduced to a cell c_f, we rewrite DOMAIN c_f as $\{t[1] : t \in \mathsf{funrel}(\mathsf{c}_f)\}$, that is, we map every pair in the relation $\mathsf{funrel}(\mathsf{c}_f)$ 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 $\mathsf{funrel}(\mathsf{c}_g)$ as follows:

$$\{\text{ITE}(p[1] = a, \langle a, r\rangle, p) : p \in \mathsf{funrel}(\mathsf{c}_f)\} \qquad (\textit{Except})$$

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 7.10.

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.

$$\frac{\dfrac{\langle \mathsf{c}_{fun}[\mathsf{c}_{arg}] \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad \mathsf{c}_{fun} \xrightarrow{1}_{\mathcal{A}} \mathsf{c}_{rel} \to_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n \qquad \langle \text{FROM } \mathsf{c}_1, \ldots, \mathsf{c}_n \text{ BY } \mathsf{c}_{ora} \mid \mathcal{A}, \mathsf{c}_{ora} : \mathsf{Int} \mid \nu \mid \Phi, 0 \le \mathsf{c}_{ora} \le n\rangle}{\langle \mathsf{c}_{pair} \mid \mathcal{A}_2 \mid \Phi_2 \mid \nu_2\rangle}}{\langle \mathsf{c}_{pair}[2] \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2, \mathit{WhenInDomain} \wedge \mathit{WhenOutsideDomain}\rangle} \text{ (FUNAPP)}$$

First, the rule (FUNAPP) introduces an integer oracle c_{ora}, which points either to a cell from $\mathsf{c}_1, \ldots, \mathsf{c}_n$ (when $1 \le \mathsf{c}_{ora} \le n$), or an arbitrary cell of proper type (when $\mathsf{c}_{ora} = 0$). Second, a cell c_{pair} is picked using the operator FROM $\mathsf{c}_1, \ldots, \mathsf{c}_n$ BY c_{ora}. This is the tuple that comprises a function argument and the respective result, so the rule (FUNAPP)

returns the function result $\mathsf{c}_{pair}[2]$. 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 $\mathsf{c}_{arg} = \mathsf{c}_i[1]$ are generated for $1 \leq i \leq n$.

$$\mathsf{c}_{ora} = i \rightarrow (\mathsf{c}_i[1] = \mathsf{c}_{arg} \wedge en\langle \mathsf{c}_{fun}, i, \mathsf{c}_i \rangle) \text{ for } 1 \leq i \leq n \qquad (\mathit{WhenInDomain})$$
$$\mathsf{c}_{ora} = 0 \rightarrow (\mathsf{c}_i[1] \neq \mathsf{c}_{arg} \vee \neg en\langle \mathsf{c}_{fun}, i, \mathsf{c}_i \rangle) \text{ for } 1 \leq i \leq n \qquad (\mathit{WhenOutsideDomain})$$

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 $\mathsf{Seq}(\tau)$ as a tuple $\langle \mathit{start}, \mathit{end}, \mathit{fun} \rangle$. The components *start* and *end* are integers that store the first index of the sequence and the index right after the end of the sequence respectively. The component *fun* is a function of type $\mathsf{Int} \rightarrow \tau$ that maps integers $1..n$ to values of type τ for some $n \geq 0$. The sequence operators maintain the invariant: $\mathit{start} \geq 1 \wedge \mathit{end} \leq n + 1$. Hence, the elements of sequence q are in the window of indices $[\mathit{start}, \mathit{end})$.

7.10 Control operators and quantifiers

Branching. The operator $\textsc{ite}(\mathsf{c}_p, \mathsf{c}_1, \mathsf{c}_2)$ returns the value of one of its branches, depending on the Boolean condition c_p. We use the $\textsc{from}\, \mathsf{c}_1, \mathsf{c}_2\, \textsc{by}\, \theta$ for $\theta \in \{1, 2\}$.

$$\frac{\langle \textsc{ite}(\mathsf{c}_p, \mathsf{c}_1, \mathsf{c}_2) \colon \tau \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad \dfrac{\langle \textsc{from}\, \mathsf{c}_1, \mathsf{c}_2\, \textsc{by}\, \theta \mid \mathcal{A}, \theta : \mathsf{Int} \mid \nu \mid \Phi, 1 \leq \theta \leq 2 \rangle}{\langle \mathsf{c}_{res} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2 \rangle}}{\langle \mathsf{c}_{res} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2, \theta = 1 \leftrightarrow \mathsf{c}_p \rangle} \ (\textsc{ite})$$

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 θ.

Assignments. An assignment $x' \in \mathsf{c}_S$ in $\textsc{Kera}^+$ specifies that a variable x' takes a value from the set S. Since any element of the set may be chosen, we use picking $\textsc{from}\, \mathsf{c}_1, \ldots, \mathsf{c}_n\, \textsc{by}\, \theta$ 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'.

$$\frac{\dfrac{\langle x' \in \mathsf{c}_S \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad \mathsf{c}_S \rightarrow_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n \quad \langle \textsc{from}\, \mathsf{c}_1, \ldots, \mathsf{c}_n\, \textsc{by}\, \theta \mid \mathcal{A}, \theta : \mathsf{Int} \mid \nu \mid \Phi, 0 \leq \theta \leq n \rangle}{\langle \mathsf{c} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2 \rangle}}{\langle \textsc{true} \mid \mathcal{A}_2 \mid \nu_2[x \mapsto \mathsf{c}] \mid \Phi_2, \mathit{EmptyAsgn} \wedge \mathit{NonEmptyAsgn} \rangle} \ (\textsc{Asgn})$$

Where

$$\theta = 0 \leftrightarrow \bigwedge_{1 \le i \le n} \neg en\langle \mathsf{c}_S, i, \mathsf{c}_i \rangle \qquad (EmptyAsgn)$$

$$\bigwedge_{1 \le i \le n} (\theta \neq i \vee en\langle \mathsf{c}_S, i, \mathsf{c}_i \rangle) \qquad (NonEmptyAsgn)$$

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 ν:

$$\frac{\langle x \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad x \in Vars}{\langle \nu(x) \mid \mathcal{A} \mid \nu \mid \Phi \rangle} \text{ (SUB)}$$

Existential Quantifiers. Quantified expressions are a fundamental building block of TLA$^+$, as well as KerA$^+$. Since we consider only finite sets, an existential quantifer can be replaced with disjunction. If the body of the quantified expression contains variable assignments, we translate $\exists x \in \mathsf{c}_S : p$ as the non-deterministic disjunction $p[\mathsf{c}_1/x] \oplus \ldots \oplus p[\mathsf{c}_n/x]$, where c_S is pointing to $\mathsf{c}_1, \ldots, \mathsf{c}_n$.

$$\frac{\langle \exists x \in \mathsf{c}_S : p \mid \mathcal{A} \mid \nu \mid \Phi \rangle \qquad \mathsf{c}_S \to_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n}{\langle p[\mathsf{c}_1/x] \oplus \ldots \oplus p[\mathsf{c}_n/x] \mid \mathcal{A} \mid \nu \mid \Phi \rangle} \text{ (EXISTS)}$$

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., $\neg\exists x \in \mathsf{c}_S$. In this case, we have to explore all possible valuations for x. 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 $\exists x \in \mathsf{c}_S : p$ is located under a universal quantifier. If this is not the case, we introduce a Skolem constant $\mathsf{c} \in \mathsf{c}_S$ and produce the expression $p[\mathsf{c}/x]$ instead of the disjunction. As expected, this optimization significantly reduces the number of SMT constraints.

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

The rule CHOOSE implements this semantics as follows. First, it rewrites the set $\{x \in S : p\}$ into a cell c_F of some type τ. Suppose that c_F points to the element cells $\mathsf{c}_1, \ldots, \mathsf{c}_n$. Second, the rule applies FROM $\mathsf{c}_1, \ldots, \mathsf{c}_n$ BY θ to pick a cell c_{res} using an oracle θ. The cell c_{res} is the result of rewriting the expression CHOOSE $x \in S : p$. To guarantee determinism of CHOOSE, for each type τ, we introduce an uninterpreted function $choose_\tau$

of sort $\mathsf{Set}(\tau) \to \tau$, and require $\mathit{choose}_\tau(\mathsf{c}_F) = \mathsf{c}_{res}$. Finally, the rewriting system instantiates lazy equality between the pairs cells c_F^1 and c_F^2, as well as the pairs $\mathit{choose}_\tau(\mathsf{c}_F^1)$ and $\mathit{choose}_\tau(\mathsf{c}_F^2)$, which are produced by rewriting of $\{x \in S : p\}$ and $\{y \in T : q\}$ in the rule Choose. Congruence of uninterpreted functions gives us the required determinism.

$$\frac{\dfrac{\dfrac{\langle \textsc{choose}\, x \in S : p \mid \mathcal{A} \mid \nu \mid \Phi\rangle}{\langle \{x \in S : p\} \mid \mathcal{A} \mid \nu \mid \Phi\rangle} \quad \mathsf{c}_F \to_{\mathcal{A}_2} \mathsf{c}_1, \ldots, \mathsf{c}_n}{\langle \textsc{from}\, \mathsf{c}_1, \ldots, \mathsf{c}_n \,\textsc{by}\, \theta \mid \mathcal{A}_2, \theta : \mathsf{Int} \mid \nu_2 \mid \Phi_2, 0 \le \theta \le n\rangle}}{\langle \mathsf{c}_{res} \mid \mathcal{A}_3 \mid \nu_3 \mid \Phi_3\rangle}}{\langle \mathsf{c}_{res} \mid \mathcal{A}_3 \mid \nu_3 \mid \Phi_3, \mathit{choose}_\tau(\mathsf{c}_F) = \mathsf{c}_{res}\rangle}\ (\textsc{Choose})$$

Non-deterministic Disjunction. This operator combines several symbolic transitions $T_1, \ldots, T_k$. In contrast to the disjunction $\vee$, 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.

$$\frac{\dfrac{\langle e_1 \oplus e_2 \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0\rangle \quad \dfrac{\langle e_i \mid \mathcal{A}_{i-1} \mid \nu_0 \mid \Phi_{i-1}\rangle}{\langle \mathsf{c}_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i\rangle}\ i = 1,2}{\langle \textsc{from}\, \nu_1(x'), \nu_2(x') \,\textsc{by}\, \theta \mid \mathcal{A}_2, \theta : \mathsf{Int} \mid \nu_0 \mid \Phi_2, \theta \in \{1,2\}\rangle}}{\langle \mathsf{c}_x \mid \mathcal{A}_3 \mid \nu_0 \mid \Phi_3\rangle}}{\langle \mathsf{c}_r \mid \mathcal{A}_3, \mathsf{c}_r : \mathsf{Bool} \mid \nu_0 \circ [x' \mapsto \mathsf{c}_x] \mid \Phi, \mathsf{c}_r \leftrightarrow \mathsf{c}_1 \vee \mathsf{c}_2, \theta = 1 \to \mathsf{c}_1, \theta = 2 \to \mathsf{c}_2\rangle}\ (\textsc{Ndc})$$

7.11 Soundness of the reduction to SMT

In this section, we define KerA$^+$ models and restrict them to finite structures. The restriction to finite structures implies that every set expression in KerA$^+$ is mapped to a finite set. Further, we present two important properties of the reduction system: termination and soundness. We 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.

Models Every satisfiable KerA$^+$ formula has a model. A *model* is a pair $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}\rangle$:

1. $\mathcal{D}$ is a *domain*. It is a disjoint union of sets $\mathcal{D}_1, \ldots, \mathcal{D}_n$, each $\mathcal{D}_i$ contains values of type τ_i.

2. $\mathcal{I}$ is an *interpretation*. It assigns values from the domain to constants and KerA$^+$ operators.

We assume that the interpretation $\mathcal{I}$ is standard, that is, it follows the standard semantics of TLA$^+$, e.g., as given by [Mer12]. As usual, we use the notation $[\![e]\!]^{\mathcal{M}}$ to denote the value of a KerA$^+$ expression in a model $\mathcal{M}$.

In our work, the specification parameters are fixed. Thus, every KerA$^+$ expression "intuitively" defines only finite values. We formalize this intuition by introducing finite structures and showing that every KerA$^+$ expression e defines a finite structure, as soon as the constants in e are interpreted as finite structures (see Proposition 10).

For a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$, a value $v \in \mathcal{D}$ 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 $f : S \to T$ 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 10. *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.*

As expected, we call a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ finite, if if every value $v \in \mathcal{D}$ is a finite structure. Finally, given a state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the reduction system, a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ is *suitable* for the state, if the expression e and the constraint Φ can be interpreted with $\mathcal{M}$.

Soundness and Termination First, we show that our reduction system always terminates:

Theorem 3. *Every sequence of ARS reductions $s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots$ is finite. In other words, the reduction process terminates.*

To prove Theorem 3, we define a partial order on KerA$^+$ expressions and show that every reduction rule produces smaller expressions.

Theorem 4 formally states the soundness of our reduction system:

Theorem 4. *Let $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ for $1 \leq i \leq m$. Assume that e_0 is a formula, that is, it has type* Bool. *The formula e_0 is satisfiable if and only if the constraint $e_m \wedge \Phi_m$ is satisfiable.*

Note that if the reduction system terminates without an error, then the terminal expression e_m in Theorem 4 is a constant. Moreover, the reductions produce constraints that are compatible with SMT solvers [BFT17]:

Proposition 11. *Let $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ for $1 \leq i \leq m$. Then, every formula Φ_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. Detailed proofs are omitted. We prove the theorem by showing that the abstract reduction system satisfies six 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 $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ 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_{before} in an expression e_{after} in a special way. In particular, a model $\mathcal{M}_{after}$ of e_{after} differs from a model $\mathcal{M}_{before}$ of e_{before} in that $\mathcal{M}_{after}$ has additional constants. Hence, we call $\mathcal{M}_{after}$ an *extended* model of $\mathcal{M}_{before}$.

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 12 ensures that all invariants 1-6 are preserved by every sequence of transitions.

Invariant 1 states that our reduction system produces only well-typed expressions:

Invariant 1. *In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, the expression e is well-typed.*

Invariant 2 gives us a relation between the arenas and the Boolean constants that are introduced for the arena edges in the constraint Φ:

Invariant 2. *In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, the following holds:*

1. *Every cell c appears in either the expression e or the formula Φ if and only if it appears in $\mathcal{A}$.*
2. *Arena $\mathcal{A}$ has an edge $(\mathsf{c}_{set}, i, \mathsf{c}_{elem})$ if and only if the formula Φ contains the constant $en\langle \mathsf{c}_{set}, i, \mathsf{c}_{elem} \rangle$.*

Invariant 3 ensures that the reduction rules produce suitable models:

Invariant 3. *Let $s_{before} \rightsquigarrow s_{after}$ be a reachable transition in the ARS, and $\mathcal{M}_{before}$ a suitable model for s_{before}. An extended structure $\mathcal{M}_{after}$ from $\mathcal{M}_{before}$ is also suitable for s_{after}.*

Invariant 4 states the arena is preserving an overapproximation of every set cell:

Invariant 4. *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its extended model. Assume that c_{set} is a set cell in the arena $\mathcal{A}$. Then, the following holds:*

1. *Assume that $\mathsf{c}_{set} \rightarrow_{\mathcal{A}} \mathsf{c}_1, \ldots, \mathsf{c}_n$, for some $n \geq 0$, and c_{set} is introduced by a rule different from (FROMSET). Then, the following holds:*

$$[\![\mathsf{c}_{set}]\!]^{\mathcal{M}} \subseteq \{[\![\mathsf{c}_1]\!]^{\mathcal{M}}, \ldots, [\![\mathsf{c}_n]\!]^{\mathcal{M}}\}$$

2. *Assume that c_{set} is a reduction of the expression FROM $\mathsf{c}_1, \ldots, \mathsf{c}_n$ BY θ with $1 \leq [\![\theta]\!]^{\mathcal{M}} \leq n$ and $\mathsf{c}_{set} \rightarrow_{\mathcal{A}} \mathsf{c}^1_{pick}, \ldots, \mathsf{c}^m_{pick}$. Then, the following holds*

$$[\![\mathsf{c}_{set}]\!]^{\mathcal{M}} \subseteq \{[\![\mathsf{c}^1_{pick}]\!]^{\mathcal{M}}, \ldots, [\![\mathsf{c}^n_{pick}]\!]^{\mathcal{M}}\}$$

Invariant 5 states that a function cell is always pointing to the associated relation cell:

Invariant 5. *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS. Assume that c_f is a function cell of type $\tau_1 \to \tau_2$ in the arena $\mathcal{A}$. Then, there is a cell c_{rel} of type $\mathsf{Set}(\tau_1 * \tau_2)$ such that the function cell is pointing to it: $\mathsf{c}_f \rightarrow_{\mathcal{A}} \mathsf{c}_{rel}$.*

Finally, Invariant 6 is about the equality between a function cell c_f in the arena and its set representation constructed based on the corresponding cell c^f_{rel}.

Invariant 6. *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its extended model. Assume that c_f is a function cell, and $\mathsf{c}_f \rightarrow_{\mathcal{A}} \mathsf{c}_{rel}$. Then, it follows that the set $[\![\mathsf{c}_{rel}]\!]^{\mathcal{M}_{after}}$ is equal to the set $[\![\{\langle x, f\,(x)\rangle : x \in \text{DOMAIN } f\}]\!]^{\mathcal{M}_{after}}$.*

The following proposition states that the above introduced invariants hold true:

Proposition 12. *Let $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ be a sequence of states produced by an abstract reduction system. Then, Invariant 3 is preserved by every transition $s_i \rightsquigarrow s_{i+1}$ for every $0 \leq s < m$. Moreover, Invariants 1–2, and 4–6 are preserved by every state s_j for every $0 \leq j \leq m$.*

7.12 Implementation

We have implemented the symbolic model checker for TLA$^+$ in Scala. It implements the stages shown in Figure 7.1, including the reduction rules introduced in Sections 7.5–7.10. 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 Inv that satisfies two conditions: (1) $Init \Rightarrow Inv$, and (2) $Inv \wedge Next \Rightarrow Inv'$. Formula Inv' is a copy of Inv, where every variable x is replaced with its primed version x'. The invariant formula Inv usually contains a constraint on the possible values of the variables such as $x \in 1..10$.

Recall that the formula $Next$ is decomposed into a non-deterministic disjunction of symbolic transitions $T_1 \oplus \ldots \oplus T_m$ in the preprocessing phase (see Section 7.3). Our model checker tests Condition (2) for each transition T_i, that is, it applies the reduction system to the initial state $\langle Inv \wedge T_i \wedge \neg Inv' \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0 \rangle$ and obtains the final state $\langle \mathsf{c}^i_{final} \mid \mathcal{A}_k \mid \nu_k \mid \Phi_k \rangle$. The tool asks the solver, whether $\Phi_k \wedge \mathsf{c}_{final^i}$ 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 (7.3)–(7.4) show a series of reductions that are used to encode an 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 ν_i to $\nu_i[\vec{x} \mapsto \vec{x}', \vec{x}' \mapsto \bot]$.

$$\langle Init' \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0 \rangle \rightsquigarrow^* \langle \mathsf{c}_1 \mid \mathcal{A}_1 \mid \nu_1 \mid \Phi_1 \rangle \tag{7.3}$$

$$\langle Next \mid \mathcal{A}_i \mid \nu_i[\vec{x} \mapsto \vec{x}', \vec{x}' \mapsto \bot] \mid \Phi_i \rangle \rightsquigarrow^* \langle \mathsf{c}_{i+1} \mid \mathcal{A}_{i+1} \mid \nu_{i+1} \mid \Phi_{i+1} \rangle \text{ for } 1 \leq i \leq k \tag{7.4}$$

To check, whether the property P can be violated after the transition $i-1$, the tool rewrites $\neg P$ as in Equation (7.5). Then, the SMT formula $\Phi_i^{\neg P} \wedge \mathsf{c}_i^{\neg P} \wedge \bigwedge_{1 \leq j \leq i} \mathsf{c}_j$ states that the property P is violated after the transition $i-1$. Satisfiability of this formula

Table 7.2: The list of TLA$^+$ benchmarks

Name	LOC	Description
Bakery-n	113	Bakery algorithm for mutual exclusion of n processes by Lamport
bcastByz-n	99	Reliable broadcast of n processes, f Byzantine faults by Srikanth & Toueg
bcastFolk-n	85	Folklore broadcast of n processes with f crash faults by Chandra et al.
EWD840-n	71	Termination detection in a ring of n processes by Dijkstra
Paxos-n	126	Paxos consensus (Synod) for n acceptors with crash faults by Lamport
Prisoners-n	75	Puzzle of n prisoners
Raft-n	363	Raft consensus for n processes and crash faults by Ongaro
SimpAlloc-c-r	68	Simple resource allocator with c clients and r resources by Merz
Traffic	32	Traffic example by [Way18]
TwoPhase-n	129	Two-phase commit with n resource managers by Gray & Lamport

gives us a counterexample.

$$\langle \neg P \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle \rightsquigarrow^* \langle \mathfrak{c}_i^{\neg P} \mid \mathcal{A}_i^{\neg P} \mid \nu_i^{\neg P} \mid \Phi_i^{\neg P} \rangle \text{ for } 1 \leq i \leq k \tag{7.5}$$

7.13 Experiments

In the following, we introduce our experiments with APALACHE and TLC that 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.

7.13.1 Benchmarks

For most of our examples, we used the benchmarks from the TLA$^+$ repository of examples [TLA20]. The traffic example is given by [Way18]. Table 7.2 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

Table 7.3: The experiments on checking inductive invariants with TLC and APALACHE.

		APALACHE					TLC		
#	Name	time	memory	#tr	#cells	#clauses	time	memory	#states
1	Bakery-5	1m33s	1.10G	16	25K	131K	-	-	-
2	EWD840-10	5s	687M	4	5.2K	36K	2s	171M	2.0K
3	bcastByz-4	3s	407M	5	1.7K	10K	2s	401M	8
4	TwoPhase-7	4s	608M	7	4.8K	23K	2h44m	2.28G	1.14M

Table 7.4: Checking candidates for inductive invariants with TLC and APALACHE that are violated.

		APALACHE					TLC		
#	Name	time	memory	#tr	#cells	#clauses	time	memory	#states
1	Bakery-5	51s	873M	16	15K	85K	-	-	-
2	EWD840-9	5s	453M	4	2.4K	19K	39s	3.35G	4.47M
3	EWD840-11	5s	482M	4	2.4K	19K	11m32s	4.41G	92M
4	EWD840-13	5s	449M	4	2.4K	19K	16h52m	5.55G	1.17B
5	bcastByz-4	4s	271M	5	463	1.1K	1s	134M	65
6	bcastByz-10	3s	298M	5	1.2K	5.4K	18m21s	3.40G	16M
7	TwoPhase-7	6s	483M	7	3.3K	16K	2h47m	2.28G	2.28M
8	TwoPhase-9	6s	642M	7	4.6K	28K	TO	2.28G	-
9	TwoPhase-11	7s	737M	7	6.0K	43K	TO	2.27G	-

complex expressions that would not be handled by our tool otherwise. We neither introduced simplifications nor 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.

7.13.2 Experiments with inductive invariants

As explained in Section 7.12, 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 7.5: The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are satisfied.

		APALACHE						TLC			
#	Name	time	memory	#tr	#cells	#clauses	depth	time	memory	#states	depth
1	Traffic	6s	221M	4	525	1.0K	4	2s	112M	4	4
2	Prisoners-4	3m19s	355M	4	2.5K	6.6K	15	1s	133M	214	14
3	Bakery-5	18ms	774M	16	14K	48K	8	-	-	-	-
4	EWD840-4	56s	1.13G	4	36K	257K	12	1s	170M	1.5K	12
5	EWD840-10	13m	1.17G	4	89K	635K	30	21m	3.40G	15M	30
6	SimpAlloc-2-2	34s	371M	3	2.9K	9.7K	7	1s	136M	64	5
7	SimpAlloc-5-3	2h56m	722M	3	5.5K	30K	7	1m49s	2.30G	1.14M	9
8	bcastFolk-4	20s	712M	4	11K	33K	10	41s	2.28G	501K	9
9	bcastFolk-20	1m09s	1.11G	4	37K	141K	10	TO	3.34G	1.14M	2
10	bcastByz-4	9m14s	1.13G	5	54K	216K	10	2s	346M	1.8K	7
11	bcastByz-6	3h00m	1.18G	5	106K	543K	11	3h42m	4.47G	15M	11
12	TwoPhase-3	1m13s	475M	7	3.0K	10K	11	1s	144M	288	11
13	TwoPhase-7	44m	516M	7	4.0K	15K	10	13s	1.13G	296K	23
14	Paxos-3	1h37m	825M	4	22K	50K	13	1m21s	2.29G	185K	25
15	Paxos-5	7h09m	1015M	4	34K	79K	14	TO	4.49G	86M	22
16	Raft-5	2h47m	1.18G	23	116K	445K	8	-	-	-	-

to check how quickly the solvers would be able to detect invariant violation as opposed to verifying the absence thereof.

Table 7.3 summarizes the results of the experiments with the original invariants, whereas Table 7.4 summarizes the results obtained when using the invalid invariant candidates. The columns "time" and "memory" show resource usage statistics, while the column "#states" shows the number of distinct states explored by TLC. Finally, the columns "#tr", "#cells", and "#clauses" display the number of symbolic transitions, the number of cells in the final arena, and the number of SMT clauses introduced by APALACHE. The abbreviation 'TO' means timeout of 23 hours.

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".

Table 7.6: The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are violated.

		APALACHE						TLC			
#	Name	time	memory	#tr	#cells	#clauses	depth	time	memory	#states	depth
1	SimpAlloc-5-3	5s	323M	3	1.7K	6.8K	4	2s	236M	6.7K	5
2	SimpAlloc-3-5	3s	315M	3	1.5K	6.2K	4	3s	679M	72K	5
3	bcastByz-4	2s	254M	5	461	989	1	1s	134M	5	2
4	bcastByz-12	49s	949M	5	20K	120K	5	3m00s	3.37G	8.89M	6
5	bcastFolklore-20	2s	301M	4	1.5K	4.9K	1	12s	2.22G	2	2
6	Paxos-3	4s	437M	4	4.7K	10K	6	2s	293M	1.0K	7
7	Prisoners-8	4s	416M	4	2.8K	9.2K	13	1s	187M	7.2K	14
8	Prisoners-10	7s	525M	4	4.2K	14K	17	3s	617M	66K	18
9	TwoPhase-5	6s	402M	7	2.2K	7.3K	9	1s	148M	436	10
10	EWD840-10	18s	753M	4	14K	55K	9	15m44s	4.41G	12M	10
11	EWD840-12	30s	824M	4	17K	68K	11	9h11m	5.53G	241M	12

7.13.3 Experiments with bounded model checking

Table 7.5 summarizes the results of the experiments with bounded model checking of safety properties. Table 7.6 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 7.3, see Section 7.13.2. 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.

7.13.4 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:

$$\begin{aligned} \text{Init} &\triangleq x = 0 \\ \text{Next} &\triangleq x' = 1 - x \lor x' = x \end{aligned}$$

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.

7.14 Related work

7.14.1 Interactive theorem provers and SMT

[MV18] introduced two encodings to translate TLA$^+$ to SMT formulas: an untyped one and a multi-sorted one. Their work is designed towards proving unsatisfiablity of

obligations inside the TLA Proof 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. In contrast, our tool supports automatic verification. We first tried to use the untyped encoding for bounded model checking, but the search space of Z3 was significantly larger even for small examples than in the case of a multi-sorted encoding. While our type system is similar to one in [MV18], our abstract reduction system applies a quantifier-free encoding, and unrolls a complete TLA$^+$ specification up to k steps. This allows us to check satisfiability (when finding enabled transitions or counterexamples) as well as unsatisfiability (when proving that a transition is disabled and an invariant holds true).

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 high-order logic, the translation meets challenges in high-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.

SMTCoq [EMT$^+$17] is a plug-in for integrating SMT and SAT solvers into the interactive theorem prover Coq [BC13]. The primary use case for SMTCoq aims at increasing the level of automation in Coq. SMTCoq 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, SMTCoq validates the certificate and generates a Coq proof for the original goal.

Several projects on proving correctness of distributed algorithms with interactive theorem provers were conducted by [HHK$^+$17], [WWP$^+$15], [RGBC17], [SWT18], [AMW18], and [vGKB$^+$19]. Although, guarantees provided by such proofs are much stronger, they demand a different level of verification efforts.

7.14.2 Semi-automated provers using decision procedures

[PLSS17] checked safety of several variants of Paxos in the effectively-propositional fragment of uninterpreted first-order logic (EPR). In their approach, the user specifies the transition system in first-order logic by means of uninterpreted relations and constants. The tool aids the user in interactive discovery of inductive invariants. Further, in order to fit the verification problem in EPR, the user has to come up with so-called derived relations. This is a powerful method that can be used for parameterized verification. However, the user has to invest more efforts in expressing the algorithms in uninterpreted first-order logic and interacting with the tool.

[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.

7.14.3 Model checkers for specialized languages

Promela is the input language of the model checker Spin [Hol03]. It supports Boolean and integer variables, arrays, processes, message channels, arithmetic, and temporal operators. Spin is an explicit-state model checker that was applied to several industrial problems. Moreover, [DTT14] checked a version of Paxos and [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 [CCG$^+$02] and nuXMV [CCD$^+$14] stem from the symbolic model checker SMV [McM93]. They are designed for modeling finite-state hardware protocols. The SMV language is much more restrictive than TLA$^+$.

Several techniques and tools for parameterized verification of fault-tolerant distributed algorithms were introduced by [DHV$^+$14, DHZ16], [FKP16], [vGBR16], [KLVW17b], and [MSB17]. The efficiency of these techniques comes from the restriction to special domains, whereas our approach applies to virtually any TLA$^+$ specification over finite structures.

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.

7.15 Conclusions

We have presented the 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.

Our work is the first step on the path towards developing an efficient symbolic model checker for TLA$^+$. Indeed, many reduction rules can be optimized for specific fragments of 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 [KNR05, BLL$^+$19]. More importantly, our framework opens the door for applying more advanced techniques such as abstraction [CGJ$^+$03, BMMR01] and reduction [Lip75, CL98]. Reductions were shown to be efficient for special classes of fault-tolerant distributed algorithms by [KLVW17b, DDMW19, vGKB$^+$19]. We are going to explore similar techniques, in order to check complex TLA$^+$ specifications of Raft by [Ong14], Disk Paxos [GL03b], and Egalitarian Paxos by [MAK13].

CHAPTER 8

Type inference for TLA$^+$: typing the untyped

The techniques described in this chapter were implemented by the author as a prototype in the Apalache model checker.

8.1 Introduction

When developing an automatic analysis tool for TLA$^+$ — be it a model checker, a static analyzer, or a theorem prover — one immediately realizes that the language poses a problem to a tool developer: *TLA$^+$ is untyped by design.* The effects of this design decision are discussed by [LP99]. In a nutshell, TLA$^+$ builds on untyped set theory by Zermelo and Fränkel and provides the user with the syntax for expressing transition systems in terms of Booleans, integers, sets, functions, records, tuples, and sequences. In the most extreme semantics of TLA$^+$, every value is simply a set. Hence, every operation is applicable to any value, though some operations produce "silly" values such as the result of $1 + \{2\}$. Another use of untyped semantics is to construct a function of integers, e.g., $[i \in 1..10 \mapsto i + 1]$, and later use sequence operations over it.

Since it is hard to develop an automatic analysis tool for a completely untyped language, the existing tools introduce a type system and run some form of type inference or type checking. TLC checks type compatibility when computing system states, e.g., it rejects a set of integers and Booleans $\{2020, \text{TRUE}\}$. TLAPS infers types of the verification conditions that are sent to an SMT solver [MV12a, MV14, MV18]. ([MV18] also introduced a purely untyped encoding.) TLA2B implements type inference similar to the one in TLAPS, whereas Apalache performs a simple top-down type propagation and falls back to user annotations, when type inference fails.

While the type systems of the tools for TLA$^+$ differ in subtle details, they have a lot in common. These type systems resemble the type system of many-sorted first-order logic — the input language of SMT solvers. This is not surprising, as at least TLAPS and Apalache use SMT solvers in the background. ProB implements constraint solving. However, in all these tools, type inference applies only to the code that went through a preprocessing phase, specific to each tool. For instance, Apalache inlines all operator definitions before running type inference.

Contributions. In this chapter, we present three major contributions:

1. We introduce a type system τ^{TLA} that encompasses the type systems of TLAPS, TLA2B, and Apalache. Moreover, our type system supports TLA$^+$ operators and thus requires no additional preprocessing.
2. We introduce a type inference technique for τ^{TLA} that translates the type inference problem to SMT constraints (in the theory of quantifier-free uninterpreted first-order formulas).
3. We implement this technique and experimentally show that only a few standard benchmarks are not well-typed in τ^{TLA} (see below).

As TLA$^+$ — by design — is a declarative specification language, rather than an imperative or functional programming language, it has the following features, which complicate type inference:

1. Instead of a clear control flow, the code imposes constraints on values.
2. The language is centered around the application of built-in operators.
3. Several built-in operators are overloaded, e.g., function application or record access.
4. Users define their own operators and functions, both allowing for recursion.

The combination of these features makes it hard to apply the classical unification algorithms, e.g., by [DM82]. From that perspective, a constraint-based approach, e.g., [Pie04][Ch. 10], seems to be a better fit. However, overloading requires us to add disjunctive constraints. Our approach to type inference is inspired by the work by [SS08], who encode the type inference problem as a problem in constraint logic programming. We are using the SMT solver Z3 [DB08]. To make our SMT encoding simple and decidable, we sacrifice the ability to compute principal types for the user-defined operators. This, however, does not seem to be an issue, if we like to use types for model checking.

We have implemented a type inference tool for TLA$^+$ in Scala and conducted experiments on 35 specifications from the repository of TLA$^+$ examples [TLA20]. Out of 35

benchmarks, only 7 specifications were not well-typed in τ^{TLA}. Four of them contained actual errors that were flagged by the type checker. The three remaining specifications used the untyped semantics. However, we easily made all three specifications well-typed by changing 2-3 lines of code. Importantly, TLA$^+$ specifications rarely have more than 2,000 lines of code. Hence, the use of SMT does not usually impact the tool performance.

Our results demonstrate that although TLA$^+$ was designed as an untyped language, the users rarely exploit the truly untyped features of the logic. We offer two explanations of this. First, the system engineers are using typed languages in their jobs, so they tend to write well-typed specifications in TLA$^+$ too. Second, as part of the design process, the users run the TLC model checker that rejects the obviously type incorrect code. (However, if TLC does not explore a state that demonstrates a type problem, it will never flag an error.)

Applications. The TLA$^+$ community will benefit from having a baseline type inference tool:

1. It will help the users to write specifications that are well-typed across different tools.
2. It opens a way to translating TLA$^+$ specifications in programming languages such as Scala, OCaml, and Rust. Executable specifications can aid in prototyping and testing of distributed systems. Without types, one has to restrict the language, e.g., as in PGo [Zha16].
3. It will boost development of the automatic analysis tools for TLA$^+$, as the absence of types hinders further research in this direction.

Structure. This chapter is organized as follows. Sections 8.2–8.3 give a brief introduction in TLA$^+$ and its normalized form. Section 8.4 introduces the type system τ^{TLA}. Section 8.5 discusses the type inference problem as a series of type inference rules. Section 8.6 presents our encoding of types in first-order logic, whereas Section 8.7 introduces instances of type schemas for built-in operators. We discuss type inference arbitrary TLA$^+$ expressions using first-order logic in Section 8.8 and demonstrate the soundness of our approach. Finally, Section 8.9 presents the experimental results. Section 8.10 discusses related work. The chapter finishes with discussions in Section 8.11 and conclusions in Section 8.12.

8.2 A refresher on TLA$^+$: Notable features

Many of our decisions regarding the type system and type inference for TLA$^+$are based on the language constructs that are not common in programming languages. We highlight such features in this section. For a comprehensive introduction to TLA$^+$, see [Mer12, Lam02].

MODULE *Paxos*

```
EXTENDS Integers
CONSTANTS Value, Acceptor, Quorum
VARIABLES msgs, maxBal, maxVBal, maxVal
...
Send(m) ≜ msgs' = msgs ∪ {m}
Phase1a(b) ≜ ∧ Send([type ↦ "1a", bal ↦ b])
             ∧ UNCHANGED ⟨⟨maxBal, maxVBal, maxVal⟩⟩
...
Phase2a(b) ≜ ∧ ¬(∃m ∈ msgs: m.type = "2a" ∧ m.bal = b)
             ∧ ∃Q ∈ Quorum:
                 LET Q1b ≜ {m ∈ msgs: m.type = "1b" ∧ m.acc ∈ Q ∧ m.bal = b}
                     Q1bv ≜ {m ∈ Q1b: m.mbal ≥ 0}
                 IN  ∧ ∀a ∈ Q: ∃m ∈ Q1b: m.acc = a
                     ∧ ∨ Q1bv = {}
                       ∨ ∃m ∈ Q1bv: ∧ m.mval = v
                                    ∧ ∀mm ∈ Q1bv: m.mbal ≥ mm.mbal
             ∧ Send([type ↦ "2a", bal ↦ b, val ↦ v])
             ∧ UNCHANGED ⟨⟨maxBal, maxVBal, maxVal⟩⟩
...
```

Figure 8.1: An excerpt from the TLA$^+$ specification of Paxos [L$^+$01]

Figure 8.1 shows an excerpt from the TLA$^+$specification of the Paxos algorithm, authored by Leslie Lamport. This specification demonstrates both the richness of TLA$^+$, as well some of the issues when it comes to designing a type system, discussed in more detail in later sections.

A TLA$^+$ specification describes a set of computations, each of them being a sequence of states. Every state assigns values to the variable names (declared with VARIABLES). The specification can be also parameterized with some values (declared with CONSTANTS). In Paxos, the parameters are: a set of values called *Value*, a set of processes called *Acceptors*, and a set of sets of processes called *Quorum*. These types are not specified, but come from our understanding of the specification.

In Paxos, the processes from the set *Acceptor* have to unanimously decide on a value from the set *Value*. As some processes may crash, the processes have to coordinate in several phases, in order to reach distributed consensus. To this end, a process starts voting by sending a unique ballot number in a message that contains the tag *"1a"*. This is captured by the operator $Phase1a(b)$, in which the set *msgs* is extended with the record $[type \mapsto \text{"1}a\text{"}, bal \mapsto b]$, for a natural number $b \in Nat$. The variable *msgs* is updated by the operator $Send(m)$. The expression $msgs'$ refers to the value of the variable *msgs* in the next state, and thus the constraint $msgs' = msgs \cup \{m\}$ extends the set with m. The other state variables must preserve their values in the next state, as indicated by the

expression UNCHANGED $\langle maxBal, \dots \rangle$. The expressions starting with "$\wedge$" are part of a multi-line conjunction.

The operator *Phase2a* contains a few complex constraints. For instance, it filters the set *msgs* with a set comprehension $\{m \in msgs : m.type = \text{"}1b\text{"} \wedge \dots\}$, and binds the expression to the nullary operator *Q1b*. Note that several expressions quantify over the sets *Q*, *Q1b*, *Q1bv*, and *Quorum*. If all constraints of *Phase2a* hold true, the set $msgs'$ becomes an extension of *msgs* with the record $[type \mapsto \text{"}2a\text{"}, bal \mapsto b, val \mapsto v]$.

Although in the text above we referred to the values as naturals, records, "sets of records", or "sets of sets", the types of variables or expressions are not apparent in TLA$^+$. Often, the parameter *Proc* is defined as a set of integers or strings, but it could be as well a set of functions or tuples, etc.

Operators. TLA$^+$ comes with plenty of built-in operators. For instance, we have seen the set operators $\cup$ and $\{\}$ as well as logic operators $\exists$, $\forall$, $\wedge$, $\vee$, and $\neg$ in Figure 8.1. It also has operators on functions (in the mathematical sense), records, tuples, control flow, and temporal behavior. A specification may also import the standard operators on naturals, integers, reals, finite sets, sequences, and bags, e.g., Paxos imports operators on integers in the first line: EXTENDS *Integers*. Almost all of TLA$^+$ syntax translates into operator definitions and operator applications. We refer to these standard operators as "*built-in*", as opposite to the *user-defined* operators such as *Phase1a*, *Phase2a*, *Q1b*, and *Q1bv* in Paxos. Interestingly, TLA$^+$ operators do not support currying, that is, operator arguments are always treated as a tuple.

The operators may be of two kinds: simple and higher-order. Simple operators have parameters that are not themselves operators, whereas higher-order operators admit simple operators as parameters. (The built-in operators contain only a few higher-order operators.) Higher order operators may not, however, take other higher-order operators as arguments. Moreover, the return type of an operator cannot be an operator. Hence, the user-defined operators in TLA$^+$ are usually understood as macros.

Example 9. *Consider the following four user-defined TLA$^+$ operators:*

$$\begin{array}{l} \textit{VARIABLE } x \\ F(p) \triangleq x \cup p \qquad G \triangleq \text{"}a\text{"} + 1 \qquad H(Q(\_), r) \triangleq Q(r) \qquad I \triangleq H(F, G) \end{array}$$

In our example, F, G and I are simple operators, whereas H is higher-order. The body of G contains a legal expression, but it evaluates to a "silly" value [Lam02][Ch. 16]. In F, we see that operators may use their parameters (i.e., p), as well as state variables (i.e., x). The operators H and I demonstrate the definition and invocation of a higher-order operator.

The language also supports nested operators in the form of LET-IN definitions:

$$F(p) \triangleq \textit{LET } G(q) \triangleq \{i \in q : i \,\%\, 2 = 0\} \textit{ IN } \{i + 1 : i \in G(p)\}$$

Operators may be recursive, as shown in the example below:

$$\begin{array}{l} \textsc{recursive}\ Fact(\_) \\ Fact(n) \triangleq \textsc{if}\ n \leq 1\ \textsc{then}\ 1\ \textsc{else}\ n * Fact(n-1) \end{array}$$

An expression $Fact(10)$ is computed immediately, without advancing a TLA$^+$ computation.

Functions. TLA$^+$ also has functions, that can be understood data structures, similar to arrays or dictionaries. The example below shows several expressions related to functions:

$$[x \in Int \mapsto x + 1] \quad [y, z \in Nat \mapsto y * z] \quad f[\{1,2,3\}] \quad [f\ \textsc{except}\ ![\{1,2\}] = [\{3\}]]$$

The first expression defines the function that increments its only integer argument. The second expression defines the function that multiplies its two natural arguments. The third expression applies a function f to its argument $\{1,2,3\}$. The fourth expression produces the function that is identical to the function f everywhere on f's domain except that the new function returns value $\{3\}$ when called with the argument $\{1,2\}$. Further, one can define the set of all functions from a set S to a set T by writing $[S \to T]$.

There is a way to define a recursive function by using a special form. For instance, the set cardinality function can be defined as follows:

$$card[S \in \textsc{subset}\ T] \triangleq \textsc{if}\ S = \{\}\ \textsc{then}\ 0\ \textsc{else}\ 1 + card[S \setminus \{\textsc{choose}\ x \in S : \textsc{true}\}]$$

In the above example, expression SUBSET T defines the powerset of T, and thus the function *card* is defined on all subsets of T. Further, the CHOOSE operator picks an element of S that satisfies a predicate, in our case, just TRUE.

To sum up, the main differences between TLA$^+$ operators and TLA$^+$ functions are as follows:

1. functions are first-class values, they can be passed everywhere, whereas operators can be either applied to their arguments, or passed in higher-order operators.
2. the operators do not have domains, whereas the functions do.
3. the operators can be polymorphic.

Further, TLA$^+$ has special syntax for records and tuples. For instance, the expressions $[id \mapsto 2020, next \mapsto \{2021, 2022\}]$ and $\langle 1, \{\{2\}\}\rangle$ define a record and a tuple, respectively. Moreover, the language contains operators on sequences. In the standard (untyped) semantics, records, tuples, and sequences are simply defined as functions over the domains of strings and integers.

Untyped features. In theory, the language does not pose any constraints on the set contents. For instance, the set $\{1, \text{“}abc\text{”}, \{2, 3\}\}$ contains three elements of different types. Same applies to function domains:

$$[p \in \{\text{“}xyz\text{”}, 21\} \mapsto \text{IF } p \in \mathit{Int} \text{ THEN “}xyz\text{” ELSE } 21] \tag{8.1}$$

The function defined in Equation (8.1) returns an integer or a string, depending on whether its argument is a string or an integer, respectively. In TLA$^+$ semantics, a function can be also applied to an arbitrary value. However, the function outcome is predictable only for the values in its domain.

There is no distinction between functions, tuples, sequences, or records. For instance, one can define a function and apply the sequence operator *Tail* to it:

$$\text{LET } f \triangleq [p \in 1..3 \mapsto p * p] \quad \text{IN} \quad \mathit{Tail}(f) \tag{8.2}$$

The sequence computed in Equation (8.2) is the sequence $\langle 4, 9\rangle$, which is also a tuple, or a function.

8.3 Normalized TLA$^+$

TLA$^+$ has rich syntax that allows the users to write specifications according to their taste. To simplify presentation, we assume that TLA$^+$ specifications are in the *normalized form*:

1. All names in the specification are unique, including the operator names, parameter names, and the names of the bound and state variables.
2. The specification has a singular operator definition, which may contain nested LET-IN definitions, including recursive operators and functions. That, is the specification has the form:

$$\begin{aligned} \mathit{Spec} \triangleq\ & \text{LET } F_1(p_1^1, \ldots, p_{n_1}^1) \triangleq \ldots \text{ IN} \\ & \ldots \\ & \text{LET } F_m(p_1^m, \ldots, p_{n_m}^m) \triangleq \ldots \text{ IN} \\ & \text{TRUE} \end{aligned}$$

Every TLA$^+$ specification can be rewritten in the normalized form by applying α-conversion and rewriting operator definitions as a chain of LET-IN definitions inside the operator *Spec*.

Table 8.1: Examples of TLA$^+$ expressions and their possible types

Expression	Type	Expression	Type
$42,\ 2+2$	Int	"abc"	String
TRUE, $2+2=4$	Bool	$3>2 \land 100<2$	Bool
$\{\{1,2\},\{3\}\}$	Set(Set(Int))	$[\{1,3\} \to \{3,5\}]$	Set(Int → Int)
SUBSET $\{1,2,3\}$	Set(Set(Int))	UNION $\{\{1\},\{2\}\}$	Set(Int)
$[x \in 1..3 \mapsto x+1]$	Int → Int	$\langle 1, \text{"abc"}\rangle$	⟨Int, String⟩
$[x \in 1..3 \mapsto x+1][2]$	Int	$\langle 1, \text{"abc"}\rangle\,[1]$	Int
DOMAIN $[x \in \{1\} \mapsto x]$	Set(Int)	$\{1,2\} \times \{\text{FALSE}\}$	Set(⟨Int, Bool⟩)
$[a \mapsto 1, b \mapsto \text{"abc"}]$	$[a \mapsto \text{Int}, b \mapsto \text{String}]$	$\emptyset$	Set(α)
$[a : \{1,2\}]$	Set($[a \mapsto \text{Int}]$)	$\langle 1\rangle$	⟨Int⟩ , Seq(Int)
$[a \mapsto 1, b \mapsto \text{"abc"}].a$	Int		

Remark. Technically, mutually-recursive operators in TLA$^+$ do not fit in our normal form. We slightly deviate from the standard syntax to simplify the presentation. Our approach works for mutually-recursive operators.

Notation. In the following, we use the notation for TLA$^+$ constructs: $F, G, H, \ldots$ for operator names (either built-in or defined with LET-IN); $p, q, r, \ldots$ for simple operator parameters (which are not operators themselves); $P, Q, R, \ldots$ for higher-order operator parameters; x, y, z for state variables; $C_1, \ldots, C_n$ for constants; and $e,\ e_1, e_2, e_3, \ldots$ for TLA$^+$ expressions.

8.4 Defining the type system τ^{tla}

We extend the type system introduced by [MV12a] as follows:

$$\begin{aligned} \tau &::= \alpha \mid \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{String} \mid \tau \to \tau \mid \mathsf{Set}(\tau) \mid \mathsf{Seq}(\tau) \mid \\ &\qquad \langle \tau, \ldots, \tau \rangle \mid \langle i_1 \mapsto \tau, \ldots, i_k \mapsto \tau \rangle \mid [h_1 \mapsto \tau, \ldots, h_k \mapsto \tau] \mid \langle \tau, \ldots, \tau \rangle \Rightarrow \tau \\ s &::= \forall \vec{\alpha}\ .\ \tau \mid s \sqcup s \end{aligned}$$

Let $\mathcal{T}^\alpha$ be the set of all types that are derived by the rule τ in the above grammar. By $\mathrm{FV}(\tau)$ we denote the set of all type variables appearing in $\tau \in \mathcal{T}^\alpha$. Then, $\mathcal{T}$ is the set of all *monotypes*, that is, $\{\tau \in \mathcal{T}^\alpha \mid \mathrm{FV}(\tau) = \emptyset\}$.

Types like Int, Set($\cdot$), and $\tau_1 \to \tau_2$ represent integers, sets, and TLA$^+$ functions, respectively. Further, we have tuples $\langle \tau_1, \ldots, \tau_k \rangle$, sparse tuples $\langle i_1 \mapsto \tau_1, \ldots, i_k \mapsto \tau_k \rangle$, records $[h_1 \mapsto \tau, \ldots, h_k \mapsto \tau]$, and operators $\langle \tau, \ldots, \tau \rangle \Rightarrow \tau$. There is no special syntax for sparse tuples in TLA$^+$, but we need them to talk about operator signatures. The role of schema types derived from s will be discussed in Section 8.4.2. Note that the operators receive a single tuple as its argument. This is due to absence of currying in TLA$^+$. The type

Table 8.2: Legal TLA$^+$ expressions that are not type correct in our system

Expression	**Why it is ill-typed**
$\{1, \text{“abc”}\}, \{1, \{2\}\}$	We allow only elements of the same type
$[a \mapsto 1, b \mapsto \text{“abc”}][x]$	If x is not a constant, it is impossible to statically find the field name that is stored in x
$\langle 1, \text{“abc”}\rangle\,[x]$	Since the types of the two fields differ, the expression $\langle 1, \text{“abc”}\rangle$ must be a tuple, not sequence. If x is a variable, it is impossible to statically find the index that is stored in x.
IF P THEN 1 ELSE TRUE	Impossible to find a unifying type for Int and Bool

Table 8.3: Examples of TLA$^+$ operators and their types

Operator	**Type**
$F \triangleq 1$	$\langle\rangle \Rightarrow \mathsf{Int}$
$G(p) \triangleq p + 1$	$\langle \mathsf{Int}\rangle \Rightarrow \mathsf{Int}$
$H(q) \triangleq \{q\}$	$\forall\alpha.\, \langle\alpha\rangle \Rightarrow \mathsf{Set}(\alpha)$
$I(J(\_), r) \triangleq J(r)$	$\forall\alpha, \beta.\, \langle\langle\alpha\rangle \Rightarrow \beta, \alpha\rangle \Rightarrow \beta$

system allows for some trivial extensions, e.g. by adding a Real type, to reason about real arithmetic. This is omitted in this chapter for brevity.

Following semantics of TLA$^+$, our approach is centered around operators and their application. Consider, for example, the set intersection, $S \cap T$, which is just the infix notation for the set operator `\intersect`(S, T). Intuitively, $\{1, 2\} \cap \{1\}$ should have the type $\mathsf{Set}(\mathsf{Int})$, whereas $\{\{1\}, \{2\}\} \cap \{\{1\}\}$ the type $\mathsf{Set}(\mathsf{Set}(\mathsf{Int}))$. However, what would be the type of $\cap$? We consider $\cap$ to be a polymorphic operator with the type $\forall\alpha.\, \langle\mathsf{Set}(\alpha), \mathsf{Set}(\alpha)\rangle \Rightarrow \mathsf{Set}(\alpha)$, where α is a type variable. In general, a type $\forall\alpha_1, \ldots, \alpha_k.\, \langle\tau, \ldots, \tau\rangle \Rightarrow \tau$ represents the type of a polymorphic operator. The case where $k = 0$ corresponds to non-polymorphic operators, in which case we simply write $\langle\tau, \ldots, \tau\rangle \Rightarrow \tau$.

Importantly, not all TLA$^+$ expressions can be assigned types in our system. The following examples show cases of typable expressions and their possible types as well as untypable expressions. Table 8.1 demonstrates types that one would intuitively prescribe to typical TLA$^+$ expressions. Table 8.2 shows examples of legal TLA$^+$ expressions that we consider untypable. As one can see, similar expressions would be rejected by statically typed languages, e.g., OCaml, Java, and Scala. Table 8.3 demonstrates typical operator definitions and types they intuitively hold.

Remark 1. *To avoid reasoning about syntactic restrictions on operators at the type level, our type system allows some operator types that are never assigned to the operators that are accepted by the TLA$^+$ parser. This is not an issue in practice, because such types do not arise as solutions to constraint problems derived from valid TLA$^+$ specifications,*

where it is syntactically impossible to construct such an operator. For example, the type $\langle \mathsf{Int} \rangle \Rightarrow (\langle \mathsf{Int} \rangle \Rightarrow \mathsf{Int})$ *is definable in our type system, but the TLA*$^+$ *syntax does not admit an operator that would have such a type.*

8.4.1 Need for subtyping: Records and sparse tuples

We intend to keep the type system simple. For example, following Lamport, in mathematics a sequence $\mathsf{Seq}(\mathsf{Int})$ is a function: $\mathsf{Int} \to \mathsf{Int}$. In contrast, we maintain a strict distinction between such types. While sequences could be seen as functions, not every function with an integer domain is a sequence.

However, record types and sparse tuples complicate the matters. Classically, one would define a subtype relation for record types, which would consider $[h_1 \mapsto \tau_1, h_2 \mapsto \tau_2]$ as a subtype of $[h_1 \mapsto \tau_1]$, under the assumption that every instance of the supertype can be safely replaced by an instance of the subtype, but not the other way around. In TLA$^+$, we must consider the opposite direction.

Our decision comes from practical examples. It also follows the subtype relation introduced by [MV12a]. Often, specification authors represent messages with records and collect messages of different kinds in a common set of messages. For example, in Paxos in Figure 8.1, the records $r_1 = [\text{type} \mapsto \text{"}1a\text{"}, \text{bal} \mapsto 2]$ and $r_2 = [\text{type} \mapsto \text{"}2a\text{"}, \text{bal} \mapsto 3, \text{val} \mapsto 1]$ are stored in the set *msgs*. Selecting an element of such set requires an additional test, for instance:

$$\text{IF } m.type = \text{"}1a\text{" THEN } m.bal \text{ ELSE } m.val$$

Set comprehension can be used similarly: $\{m \in msgs : m.type = \text{"}1b\text{"} \wedge m.acc \in Q \ldots\}$.

Hence, TLA$^+$ records are similar to unions in C, rather than structs in C. Moreover, there are no fixed patterns for matching and unpacking records of different types. If m would be assigned the classic supertype of r_1, r_2 and r_3, namely $[\text{type} \mapsto \mathsf{String}, \text{bal} \mapsto \mathsf{Int}]$, performing access to either $m.acc$ or $m.val$ fields should trigger an error, as the record supertype has no *acc* or *val* field. In our system, the supertype of r_1 and r_2 is therefore defined to be $[\text{type} \mapsto \mathsf{String}, \text{bal} \mapsto \mathsf{Int}, \text{val} \mapsto \mathsf{Int}]$. In general, we consider the supertype of two records $[a_1 \mapsto \tau_1, \ldots, a_k \mapsto \tau_k, c_1 \mapsto \tau''_1, \ldots, c_n \mapsto \tau''_n]$ and $[b_1 \mapsto \tau'_1, \ldots, b_l \mapsto \tau'_l, c_1 \mapsto \tau''_1, \ldots, c_n \mapsto \tau''_n]$ to be $[a_1 \mapsto \tau_1, \ldots, a_k \mapsto \tau_k, b_1 \mapsto \tau'_1, \ldots, b_l \mapsto \tau'_l, c_1 \mapsto \tau''_1, \ldots, c_n \mapsto \tau''_n]$. In other words, two records have a common supertype only if they match on the types of all overlapping fields.

This approach comes with a trade-off. We sacrifice strict safety of record access. If, for example, the author had forgotten to perform a run-time check in the previous example and simply wrote:

$$\exists m \in S \,.\, x' = m.a$$

our type inference would consider the field access as type-safe, even though S could contain a record with no a field. We see this as a necessary sacrifice, if we want our type system

Table 8.4: Examples of TLA$^+$ operators and their subtypes

τ_1	τ_2	**Is $\tau_1 \triangleleft \tau_2$?**
Bool	Int	✗
$\langle 2 \mapsto \mathsf{Int} \rangle$	$\langle \mathsf{Bool}, \mathsf{Int} \rangle$	✓
$\langle 2 \mapsto \mathsf{Bool} \rangle$	$\langle \mathsf{Bool}, \mathsf{Int} \rangle$	✗
$[h_1 \mapsto \mathsf{Int}, h_2 \mapsto \mathsf{Bool}]$	$[h_1 \mapsto \mathsf{Int}, h_3 \mapsto \mathsf{Set}(\mathsf{Int})]$	✗
$\mathsf{Set}([h_1 \mapsto \langle 2 \mapsto \mathsf{Int} \rangle])$	$\mathsf{Set}([h_1 \mapsto \langle \mathsf{Int}, \mathsf{Int} \rangle, h_2 \mapsto \mathsf{Set}(\mathsf{Int})])$	✓

to be compatible with existing TLA$^+$ specifications. The issues mostly boil down to the absence of any kind of casting operator (since TLA$^+$ is untyped by design). A potential alternative solution to this kind of problem would be to introduce a standard module with a casting operator, but this would shift a lot of the work to specification authors, as the pattern described above is incredibly common in specifications of message-passing systems.

We formally define a *reflexive* and *transitive* relation $\tau_1 \triangleleft \tau_2$ for $\tau_1, \tau_2 \in \mathcal{T}$ (i.e., τ_1 is a *subtype* of τ_2) that holds, when one of the following conditions is met:

- If τ_1 is a set type $\mathsf{Set}(\tau_1')$, and τ_2 is a set type $\mathsf{Set}(\tau_2')$, as well as $\tau_1' \triangleleft \tau_2'$.
- Similar for sequences, functions, operators and fixed-size tuples.
- If τ_1 is a record type $[h_1 \mapsto \tau_1^1, \ldots, h_k \mapsto \tau_1^k]$ and τ_2 is a record type $[h_1 \mapsto \tau_2^1, \ldots, h_k \mapsto \tau_2^k, \ldots, h_{k+m} \mapsto \tau_2^{k+m}]$, as well as $\tau_1^1 \triangleleft \tau_2^1, \ldots, \tau_1^k \triangleleft \tau_2^k$, for some $m \geq 0$
- Similar for sparse tuples.
- If τ_1 is a sparse tuple type $\left\langle i_1 \mapsto \tau_1^1, \ldots, i_k \mapsto \tau_1^k \right\rangle$ and τ_2 a tuple type $\langle \tau_2^1, \ldots, \tau_2^m \rangle$, as well as $m \geq \max(i_1, \ldots, i_k)$ and $\tau_1^1 \triangleleft \tau_2^{i_1}$, $\ldots$, $\tau_1^k \triangleleft \tau_2^{i_k}$.

The only types with nontrivial supertypes are sparse tuples and records, where their supertypes are permitted to have more indices/fields, but they must agree on all shared indices/fields. In particular, every sparse tuple is a subtype of infinitely many tuples of fixed size. It is easy to see that this relation is both reflexive and transitive.

Example 10. *Table 8.4 shows instances of the subtyping relation* $\triangleleft$.

8.4.2 Annotating the built-in operators

The cornerstone of our approach is reusing the knowledge about built-in operators. By knowing their semantics in TLA$^+$, we annotate the built-in operators with (principal) type schemas. For instance, the operator for unary minus $-(x)$ in the *Integers* module has the type $\mathsf{Int} \Rightarrow \mathsf{Int}$. Likewise, logical implication $\Rightarrow$ has the type $\langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$. To annotate all operators, we have to address several issues:

1. A few built-in operators are *variadic*, they accept variable number of arguments. For instance, the set constructor $\{\dots\}$ accepts zero or more arguments.
2. Most of built-in operators are *polymorphic*, they are parameterized by one or more types. For instance, the set operators $\cup$, $\cap$, $\setminus$ accept two sets of some type α and return a set of type α.
3. A few operators are *overloaded*: application $[\_]$ and DOMAIN $\_$ that apply to functions, records, tuples, and sequences; constructor $\langle\dots\rangle$ that produces a tuple or a sequence.

The properties (1)-(3) are not mutually exclusive: the set constructor is both variadic and polymorphic, whereas the tuple/sequence constructor is variadic, polymorphic, and overloaded. Luckily, overloading is limited to several built-in operators, so there is a small range of operators to choose from. Hence, we think of overloaded operators as finite collections of polymorphic operators. For every variadic operator F, we introduce a family of fixed-arity operators $F_0, F_1(\_), F_2(\_,\_), \dots$. Since TLA$^+$ does not have currying, the operator arity is clear from the syntax in every concrete application. For instance, in the expression $\{1, 2, 3\}$, the set constructor has arity of three.

Hence, we assign to every polymorphic operator of fixed arity a parameterized type schema:

$$\forall \alpha_1, \dots, \alpha_k.\ \langle \omega_1, \dots, \omega_n \rangle \Rightarrow \hat{\omega} \tag{8.3}$$

In (8.3), the types $\omega_1, \dots, \omega_n, \hat{\omega}$ may refer to the type parameters $\alpha_1, \dots, \alpha_k$. For example:

$$+ \colon \langle \mathsf{Int}, \mathsf{Int} \rangle \Rightarrow \mathsf{Int} \qquad\qquad \cup \colon \forall \alpha.\ \langle \mathsf{Set}(\alpha), \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Set}(\alpha)$$

The variadic operators have one schema per arity. For instance, the set constructor:

$$\{\underbrace{\_, \dots, \_}_{n \text{ times}}\}_n \colon \forall \alpha.\ \underbrace{\langle \alpha, \dots, \alpha \rangle}_{n \text{ times}} \Rightarrow \mathsf{Set}(\alpha)$$

Finally, the overloaded operators get assigned a finite number of schemas that are joined with $\sqcup$. For instance, the ternary tuple/sequence constructor is assigned the type schema:

$$\langle \_, \_, \_ \rangle_3 : \left(\forall \alpha_1, \alpha_2, \alpha_3.\ \langle \alpha_1, \alpha_2, \alpha_3 \rangle \Rightarrow \langle \alpha_1, \alpha_2, \alpha_3 \rangle\right) \ \sqcup\ \left(\forall \alpha_4.\ \langle \alpha_4, \alpha_4, \alpha_4 \rangle \Rightarrow \mathsf{Seq}(\alpha_4)\right)$$

When $n = 1$, we call the schema *primitive*. Otherwise, the schema is *complex*. Appendix 8.13 lists all operator schemas we consider.

Notation. In rest of the chapter, we use the following type-related notation: τ, ρ, ω for the types derived from τ in the grammar of τ^{TLA}; notation s, $s_1, s_2, \ldots$ for the schemas derived from s; notation $\alpha, \beta, \ldots$ for type variables.

8.5 Assigning meaningful types to TLA$^+$ expressions

Before we can talk about automatic type inference, we have to understand which types may be in principle assigned to TLA$^+$ expressions. As discussed in Section 8.4.2, the built-in operators are assigned type schemas, which we define manually once and for ever.

In this section, we define a system of rules that define which types can be assigned to a TLA$^+$ expression. Some of our rules are non-deterministic, as dictated by overloaded operators, e.g., function application. Our type inference problem consists of finding at least one type assignment that follows the inference rules.

Let us fix a TLA$^+$ specification and define the set Names that includes:

- the names of constants and state variables,
- the names of bound variables, e.g., defined with $\exists x \in S : P$,
- the names of user-defined operators and their parameters, and
- the names of the built-in operators that appear in the specification.

We define a *type environment* Γ as a partial function from Names to $\mathcal{T}$. Recall that the types in $\mathcal{T}$ do not contain type variables. When Γ is undefined on $x \in \text{Names}$, we write $\Gamma(x) = \bot$. Given a type $\tau \in \mathcal{T}$ and an identifier y, we write $\Gamma, y : \tau$ to refer to the environment Γ' that extends Γ with y assigned the type τ. Formally, $\Gamma'(y) = \tau$ and $\Gamma'(z) = \Gamma(z)$, if $z \in \text{Names} \setminus \{y\}$.

Given an environment Γ and a TLA$^+$ expression e over the set Names, we write $\Gamma \vdash e : \tau$ to denote that e can be assigned the type τ by following the inference rules in the environment Γ. We define the relation $\vdash$ with a set of rules that are summarized in Table 8.2. We omit the rules for: universal quantifiers, bounded/unbounded Choose, set filtering, and function constructors. They all introduce fresh variable bindings, similar to (Exists) and (Map). However, these operators bring no additional insight about types.

The rules (Int), (Bool), and (Str) simply infer types of constant expressions. The rule (Env) retrieves the type of a name from the environment, whereas the rule (Sub) uses the subtyping relation. The most complex rules are related to operator definition and application. The rule (Let) assigns a monotype to an operator F, so that F's parameters have the types compatible with the operator body e_1. The rule (App) uses the type of F to infer its result, provided that the parameters of F agree with the actual arguments of F. The rule (LetRec) is a recursive version of the rule (Let), which

additionally assumes that the type of the result in the operator body is stable under operator application. (The TLA$^+$ syntax for recursive let-definitions is quite verbose.)

Finally, the rule (BuiltInOp) instantiates the schema of a built-in operator F with some monotype. If a schema has the complex type that includes a primitive schema $\forall\alpha_1,\ldots,\alpha_k.\ \langle\omega_1,\ldots,\omega_n\rangle \Rightarrow \hat{\omega}$, then F may be used with the type $(\langle\omega_1,\ldots,\omega_n\rangle \Rightarrow \hat{\omega})[\tau_1/\alpha_1,\ldots,\tau_k/\alpha_k]$. This type is obtained from the primitive schema by replacing the type parameters $\alpha_1,\ldots,\alpha_n$ with monotypes $\tau_1,\ldots,\tau_k$.

Because of operator overloading (e.g., tuple constructor), it is possible to assign different types to the same expression within the same environment. Consider the tuple $\langle 1,2\rangle$. Because the operator $\langle \_,\_\rangle_2$ is annotated with a complex schema, we can follow different branches of $\sqcup$ in the rule (BuiltInOp), and thus assign either the type $\langle \mathsf{Int},\mathsf{Int}\rangle$, or the type $\mathsf{Seq}(\mathsf{Int})$ to the tuple $\langle 1,2\rangle$. This non-determinism is built-in into the type system. Instead of having a unique type assignment, we may have multiple reasonable type assignments. This is a consequence of distinguishing tuples and records from functions.

Note that the relation $\vdash$ is merely a theoretical tool for us. Our techniques do not attempt to explicitly compute types using these rules, but rather construct type values in a way that guarantees their compliance with the relation $\vdash$.

Example 11. *Consider the expression $x = 1$ and the environment $\Gamma = [x \mapsto \mathsf{Int}]$. Then $\Gamma \vdash 1\colon \mathsf{Int}$ by (Int), and $\Gamma \vdash x\colon \mathsf{Int}$ by (Env). The operator "=" is annotated with the primitive schema $\forall\alpha_1\ .\ \langle\alpha_1,\alpha_1\rangle \Rightarrow \mathsf{Bool}$, so we can derive $\Gamma \vdash$ " = ": $\langle \mathsf{Int},\mathsf{Int}\rangle \Rightarrow \mathsf{Bool}$ by choosing $\tau_1 = \mathsf{Int}$ for α_1 and following the rule (BuiltInOp). Combined with the rule (App), this gives us the expected result that $\Gamma \vdash x = 1\colon \mathsf{Bool}$. Alternatively, if $\Gamma = [x \mapsto \mathsf{String}]$, then there is no type derivation for $x = 1$ in the context Γ.*

Having defined the derivation rules for $\vdash$, we are now at the position to formulate what it means to assign a type to a TLA$^+$ expression.

Definition 6. *We say that a TLA$^+$ expression e is* typable *in a environment Γ, if there is exists a type $\tau \in \mathcal{T}$, for which we can derive $\Gamma \vdash e\colon \tau$.*

In practice, given a TLA$^+$ specification e, our goal is to find a type context Γ with the properties:

- e is typable in Γ, and
- Γ is defined on the state variables (declared with VARIABLES) and their primed versions as well as specification parameters (declared with CONSTANTS).

This ensures that every expression in the specification can be assigned at least one type.

We tackle our goal in three stages. First, in Section 8.6, we show how to encode a single type in FOL, as well as how to recover types from FOL models. Second, in Section 8.7,

$$\frac{i \text{ is an integer literal}}{\Gamma \vdash i\colon \mathsf{Int}}\ (\textsc{Int}) \qquad \frac{b \text{ is } \textsc{false} \text{ or } \textsc{true}}{\Gamma \vdash b\colon \mathsf{Bool}}\ (\textsc{Bool}) \qquad \frac{s \text{ is a string}}{\Gamma \vdash s\colon \mathsf{String}}\ (\textsc{Str})$$

$$\frac{x \in \mathsf{Names} \qquad \Gamma(x) \neq \bot}{\Gamma \vdash x\colon \Gamma(x)}\ (\textsc{Env}) \qquad \frac{\Gamma \vdash e\colon \tau_1 \qquad \tau_1 \triangleleft \tau_2}{\Gamma \vdash e\colon \tau_2}\ (\textsc{Sub})$$

$$\frac{\Gamma, p_1\colon \rho_1, \ldots, p_n\colon \rho_n \vdash e_1\colon \tau_1 \qquad \Gamma, F\colon \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \tau_1 \vdash e_2\colon \tau_2}{\Gamma \vdash \textsc{let}\ F(p_1, \ldots, p_n) \triangleq e_1\ \textsc{in}\ e_2\colon \tau_2}\ (\textsc{Let})$$

$$\frac{\begin{array}{c}\Gamma, p_1\colon \rho_1, \ldots, p_n\colon \rho_n, F\colon \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \tau_1 \vdash e_1\colon \tau_1 \\ \Gamma, F\colon \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \tau_1 \vdash e_2\colon \tau_2\end{array}}{\Gamma \vdash\ \textsc{let recursive}\ F(\_, \ldots, \_)\ F(p_1, \ldots, p_n) \triangleq e_1\ \textsc{in}\ e_2\colon \tau_2}\ (\textsc{LetRec})$$

$$\frac{\tau_1, \ldots, \tau_k \in \mathcal{T} \qquad F \text{ has schema } (\forall \alpha_1, \ldots, \alpha_k.\ \langle \omega_1, \ldots, \omega_n \rangle \Rightarrow \hat{\omega}) \sqcup \ldots}{\Gamma \vdash F\colon (\langle \omega_1, \ldots, \omega_n \rangle \Rightarrow \hat{\omega})[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]}\ (\textsc{BuiltInOp})$$

$$\frac{\Gamma \vdash F\colon \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \tau \qquad \Gamma \vdash e_1\colon \rho_1 \qquad \ldots \qquad \Gamma \vdash e_n\colon \rho_n}{\Gamma \vdash F(e_1, \ldots, e_n)\colon \tau}\ (\textsc{App})$$

$$\frac{\Gamma \vdash S\colon \mathsf{Set}(\tau) \qquad \Gamma, x\colon \tau \vdash P\colon \mathsf{Bool}}{\Gamma \vdash (\exists x \in S\colon P)\colon \mathsf{Bool}}\ (\textsc{Exists})$$

$$\frac{\Gamma \vdash S\colon \mathsf{Set}(\tau_1) \qquad \Gamma, x\colon \tau_1 \vdash e\colon \tau_2}{\Gamma \vdash \{e\colon x \in S\}\colon \mathsf{Set}(\tau_2)}\ (\textsc{Map})$$

Figure 8.2: Assigning types to an expression e from an environment Γ, that is, $\Gamma \vdash x\colon S$

```
1  \* Example 1:
2  VARIABLE S
3  Init ≜ S = { { } }
4  Next ≜ ∨ ∃y ∈ S: S' = S \ {y}
5         ∨ ∃T ∈ S: ∃z ∈ 1..100:
6             S' = (S \ {T}) ∧ z ∈ T
7
8
```

```
1  \* Example 2:
2  VARIABLE S
3  Init ≜ S = { }
4  Next ≜
5    ∨ ∧ S' = S
6      ∧ ∀f, g ∈ S:
7          DOMAIN f = DOMAIN g
8    ∨ ∃n ∈ 1..10: S' = S ∪ {⟨n⟩}
```

Figure 8.3: Examples motivating the choice of a logical encoding

we show how to encode schema instances in FOL. Third, in Section 8.8, we demonstrate how to encode the type of a given TLA$^+$ expression in FOL. We can then use the third step to create constraints for the entire specification, or for an arbitrary subset, and solve them using a variety of approaches. Concretely, we opt to use an SMT solver, as the constraints lie in a decidable theory of SMT.

8.6 A logical encoding of types

In this section, we give an overview of the logic we are using for type inference with SMT. To avoid SMT specifics, we present a logic-based framework.

Example 1 in Figure 8.3 exemplifies the difficulties when applying the classical approaches such as by [DM82]. By analyzing the operator *Init*, we see that the (global) state variable S is a set of sets. However, S contains the empty set $\{\}$, so we can only conclude that S has the type $\mathsf{Set}(\mathsf{Set}(\alpha))$ for some type variable α. When analyzing the operator *Next*, we see that the first disjunct in line 4 is consistent with the type $S : \mathsf{Set}(\mathsf{Set}(\alpha))$, but it does not reveal anything new about the type of S. The second disjunct in lines 5–6 lets us clarify that the type of S is $\mathsf{Set}(\mathsf{Set}(\mathsf{Int}))$. Importantly, the type information should propagate in the both directions: (1) with the code flow, e.g., the type $z : \mathsf{Int}$ propagates in the expression under the quantifier in line 6, and (2) against the code flow, e.g., the expression $z \in T$ gives us the type $T : \mathsf{Set}(\mathsf{Int})$ and then we learn that S has the type $\mathsf{Set}(\mathsf{Set}(\mathsf{Int}))$. To deal with this example, we would have to collect a set of type constraints and elaborate them, as unification finds more precise types.

Example 2 in Figure 8.3 exemplifies the difficulties when applying the constraint-based approaches such as by [Rém92, GS01]. From line 7, we conclude that S is a set of function-like objects that have the Domain operator: TLA$^+$ functions, records, tuples, and sequences. From line 8, we conclude that S is a set of either tuples or sequences, due to the constructor $\langle \_ \rangle_1$. As a result, operator overloading produces disjunctive constraints, which are usually avoided in constraint-based approaches. Possibly, a more elaborate subtyping relation would help, but we prefer keeping the type system simple.

Table 8.5: Examples of types encoded in first-order (FO) logic

Value	Type	FO term	Value	Type	FO term
1	Int	int	$\langle 1, \text{TRUE} \rangle$	$\langle \mathsf{Int}, \mathsf{Bool} \rangle$	$\text{tup}(2, \text{int}, \text{bool}, t_3, \ldots, t_{\mathsf{NI}})$
$\{1, 2\}$	Set(Int)	set(int)	$\langle 1, 2 \rangle$	Seq(Int)	seq(Int)

Motivated by the above examples, we find that an encoding in first-order logic (FOL) better suited for our goal. It also frees us from the burden of untested constraint solver, as there are many mature off-the-shelf SMT solvers designed to tackle constraint solving in various fragments of FOL.

8.6.1 Terms for τ^{tla} in first-order logic

In the following, we fix a TLA$^+$ specification in normalized form. From this specification, we extract the finite set of tuple indices *TupInd* and the finite set of record fields *RecFld*. Denote by NI the maximal index in *TupInd* and by NF the cardinality of *RecFld*. These numbers define the signatures of the tuple and record constructors (for a fixed specification). We fix a global enumeration of record fields, i.e., a bijection $\mathsf{fld}\colon \{1, \ldots, \mathsf{NF}\} \to \mathit{RecFld}$.

First-order signatures. We use a sorted encoding, where the sort of types is denoted by S_T. Terms are constructed from the following function symbols, representing type constructors, with arities as indicated:

$\text{int}\colon S_T$
$\text{bool}\colon S_T$
$\text{str}\colon S_T$
$\text{oper}_1\colon S_T \times S_T \to S_T$
$\text{set}\colon S_T \to S_T$
$\text{seq}\colon S_T \to S_T$
$\text{tup}\colon \mathbb{Z} \times S_T^{\mathsf{NI}} \to S_T$
$\ldots$
$\text{rec}\colon S_T^{\mathsf{NF}} \to S_T$
$\text{fun}\colon S_T \times S_T \to S_T$
$\text{oper}_0\colon S_T \to S_T$
$\text{oper}_{\mathsf{NI}}\colon S_T^{\mathsf{NI}} \times S_T \to S_T$

Records and tuples need discussion. In TLA$^+$, a record constructor may introduce an arbitrary number of fields. By having the upper bound NF on the number of fields in the specification, we simply encode each record with the single function symbol rec as a record with NF fields. We let the solver pick arbitrary values for the unrestricted fields. In case of tuples, the first argument of tup specifies the tuple size, whereas the other NI arguments specify the tuple fields. Like in records, some of the fields may be unconstrained, which naturally allows us to encode sparse tuples.

Example 12. *Table 8.5 shows a few types and their encodings.*

Remark 2. *In the implementation, we do a more precise analysis by omitting those fields for which post-processing produces no constraint. This reduces the number of spurious fields. In the interest of presentation, we omit these optimizations.*

Term-Algebraic Interpretations. Not every FOL interpretation of the above is useful to us. Specifically, we say a interpretation is term-algebraic, if the universe of S_T is the term algebra defined by the function symbols int, str, bool, set, seq, tup, rec, fun, and oper, that is, the smallest set for which:

- The constants int, str and bool belong to the universe of S_T.
- If a, b belong to the universe of S_T, then so do $\mathrm{set}(a)$, $\mathrm{seq}(b)$, and $\mathrm{fun}(a, b)$.
- If $a_1, \ldots, a_{\mathsf{NI}}, b$ belong to the universe of S_T and k is an integer from $\{0, \ldots, \mathsf{NI}\}$, then $\mathrm{tup}(k, a_1, \ldots, a_{\mathsf{NI}})$ and $\mathrm{oper}_k(a_1, \ldots, a_k, b)$ belong to the universe of S_T.
- If $a_1, \ldots, a_{\mathsf{NF}}$ belong to the universe of S_T and k is an integer from $\{1, \ldots, \mathsf{NF}\}$, then $\mathrm{rec}(a_1, \ldots, a_{\mathsf{NF}})$ belongs to the universe of S_T.
- Two terms t_1 and t_2 are equal, $t_1 = t_2$, if and only if t_1 and t_2 are syntactically equal.

In the remainder of the chapter we consider only term-algebraic interpretations.

Remark 3. *For simplicity, we are using integers as arguments to* tup. *These integers are bounded by* NI. *Instead of having integers, we can introduce a fixed number of constants and a linear ordering on them. This will make our encoding fit into quantifier-free theory with uninterpreted functions, which is decidable and treated more efficiently by the SMT solvers.*

Notation. In the following, we are using the notation: a, b, c, v for first-order free variables and t for first-order terms.

8.6.2 From first-order terms to types and back

Consider an arbitrary (term-algebraic) interpretation $\mathcal{M}$. It naturally defines a mapping $\tau_{\mathcal{M}}$ of terms into monotypes (see Table 8.6).

The other direction is not as simple. If we did not have records and tuples, our translation to FOL would be just as straightforward: take a type and translate it into the corresponding first-order term. If we wanted to have a one-to-one correspondence between types and terms, we would have to introduce multiple record and tuple constructors as well as the subtype relation $\triangleleft$ in logic. *A direct definition of the subtype relation in logic requires a (quantified) axiom, which would immediately drive us outside of the quantifier-free logic.* We avoid this direct approach and combine the translation of individual types into logic with the subtype relation. Instead of keeping the record and tuple structure precise at the logic level, we essentially encode their supertypes. In the implementation, we reduce the number of fields in the post-processing step.

We seek to define *type-term compatibility* : a logic formula $[\![\tau]\!]_{\blacktriangleleft t}$, which establishes a relation between a term t and a type τ and has the following property:

Table 8.6: Constructing Types from Terms via $\tau_{\mathcal{M}}$

Term	**Type**
int	Int
bool	Bool
str	String
$\mathrm{set}(t)$	$\mathsf{Set}(\tau_{\mathcal{M}}(t))$
$\mathrm{tup}\,(k, t_1, \ldots, t_{\mathsf{NI}})$	$\langle \tau_{\mathcal{M}}(t_1), \ldots, \tau_{\mathcal{M}}(t_k) \rangle$
$\mathrm{seq}(t)$	$\mathsf{Seq}(\tau_{\mathcal{M}}(\mathsf{t}))$
$\mathrm{oper}_k(t_1, \ldots, t_k, \hat{t})$	$\langle \tau_{\mathcal{M}}(t_1), \ldots, \tau_{\mathcal{M}}(t_k) \rangle \Rightarrow \tau_{\mathcal{M}}(\hat{t})$
$\mathrm{fun}(t_1, t_2)$	$\tau_{\mathcal{M}}(t_1) \to \tau_{\mathcal{M}}(t_2)$
$\mathrm{rec}\,(t_1, \ldots, t_{\mathsf{NF}})$	$[\mathsf{fld}(1) \mapsto \tau_{\mathcal{M}}(t_1), \ldots, \mathsf{fld}(\mathsf{NF}) \mapsto \tau_{\mathcal{M}}(t_{\mathsf{NF}})]$
x (variable)	$\tau_{\mathcal{M}}(\mathcal{M}(x))$

Remark 1. *For every monotype τ, term t and model $\mathcal{M}$ of $[\![\tau]\!]_{\blacktriangleleft t}$, that is, $\mathcal{M} \models [\![\tau]\!]_{\blacktriangleleft t}$, the following holds: $\tau \triangleleft \tau_{\mathcal{M}}(t)$*

8.6.3 Defining type-term compatibility

Before giving a precise definition of $[\![\tau]\!]_{\blacktriangleleft t}$ we have to establish a relation between type variables and first-order variables. We assume that every type variable α is associated with a unique logic variable $\lceil \alpha \rceil$.

We define the type-term compatibility $[\![\tau]\!]_{\blacktriangleleft t}$ for an arbitrary term t and a type $\tau \in \mathcal{T}^{\alpha}$ recursively on the structure of τ in Table 8.7. The constraints for integers, Booleans, strings are just term equalities. The constraints for sets, sequences, functions, and operators introduce fresh logic variables (denoted with $\mathsf{fresh}(c)$). Hence, the question of type-term compatibility is propagated into the element types (respectively, argument types).

Tuples, sparse tuples, and records are the least trivial. For each record, we introduce NF fresh variables, to capture all potential fields. However, we only constrain those variables that originate from the fields $h_1, \ldots, h_n$ of the provided record types. (The sequence $i(1), \ldots, i(n)$ enumerates the indices of the fields $h_1, \ldots, h_n$ as defined by the mapping fld). Likewise for tuples, the variables that originate from the fields $1, \ldots, n$, are constrained using the types $\tau_1, \ldots, \tau_n$. As the tuple size is known, the constant n is used directly in the functional symbol tup. Finally, for the sparse tuples, the terms for the fields $i_1, \ldots, i_n$ are restricted with the constraints that originate from the types $\tau_1, \ldots, \tau_n$. However, unlike regular tuples, the size of a sparse is not fixed, but it is bound from below with the largest index appearing in the type, that is, $\max(i_1, \ldots, i_n)$. Recall that a tuple of k elements is never a subtype of a tuple of $m > k$ elements, but a sparse tuple is a subtype of an infinite number of fixed-size tuples (see Section 8.4.1).

Example 13. *Consider the (parameterized) type $\mathsf{Set}(\beta)$. By applying the rule for set*

Table 8.7: The Definition of $[\![\tau]\!]_{\blacktriangleleft t}$.

Type τ	Term-type constraint $[\![\tau]\!]_{\blacktriangleleft t}$	Comments
Int	$t = \text{int}$	
Bool	$t = \text{bool}$	
String	$t = \text{str}$	
α	$t = \lceil \alpha \rceil$	
$\mathsf{Set}(\tau_{elem})$	$t = \text{set}(c) \wedge [\![\tau_{elem}]\!]_{\blacktriangleleft c}$	$\mathsf{fresh}(c)$
$\mathsf{Seq}(\tau_{\mathsf{elem}})$	$t = \text{seq}(c) \wedge [\![\tau_{elem}]\!]_{\blacktriangleleft c}$	$\mathsf{fresh}(c)$
$\tau_1 \to \tau_2$	$t = \text{fun}(c_1, c_2) \wedge [\![\tau_1]\!]_{\blacktriangleleft c_1} \wedge [\![\tau_2]\!]_{\blacktriangleleft c_2}$	$\mathsf{fresh}(c_1, c_2)$
$\langle \tau_1, \ldots, \tau_n \rangle \Rightarrow \hat{\tau}$	$t = \text{oper}_n(c_1, \ldots, c_n, \hat{c}) \wedge [\![\tau_1]\!]_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![\tau_n]\!]_{\blacktriangleleft c_n} \wedge [\![\hat{\tau}]\!]_{\blacktriangleleft \hat{c}}$	$\mathsf{fresh}(c_1, \ldots, c_n, \hat{c})$
$[h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n]$	$t = \text{rec}(c_1, \ldots, c_{\mathsf{NF}}) \wedge \bigwedge_{j=1}^{n} [\![\tau_j]\!]_{\blacktriangleleft c_{i(j)}}$	$\mathsf{fresh}(c_1, \ldots, c_{\mathsf{NF}})$, and $\mathsf{fld}(i(j)) = h_j$ for $1 \leq j \leq n$
$\langle \tau_1, \ldots, \tau_n \rangle$	$t = \text{tup}(n, c_1, \ldots, c_{\mathsf{NI}}) \wedge \bigwedge_{j=1}^{n} [\![\tau_j]\!]_{\blacktriangleleft c_j}$	$\mathsf{fresh}(c_1, \ldots, c_{\mathsf{NI}})$
$\langle i_1 \mapsto \tau_1, \ldots, i_n \mapsto \tau_n \rangle$	$m \leq k \leq \mathsf{NI} \wedge t = \text{tup}(k, c_1, \ldots, c_{\mathsf{NI}}) \wedge \bigwedge_{j=1}^{n} [\![\tau_j]\!]_{\blacktriangleleft c_{i_j}}$	$\mathsf{fresh}(k, c_1, \ldots, c_{\mathsf{NI}})$, and $m = \max(i_1, \ldots, i_n)$

types in Table 8.7, we see that $[\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft t}$ *is as follows:*

$$t = \text{set}\,(c) \wedge [\![\beta]\!]_{\blacktriangleleft c} \qquad \textit{for some fresh}(c)$$

By applying the rule for type variables $[\![\beta]\!]_{\blacktriangleleft c}$*, we get* $\lceil \beta \rceil$*, which corresponds to some logic constant* b*. Hence, the logic constraints for* $[\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft t}$ *are as follows:*

$$t = \text{set}\,(c) \wedge c = b \tag{8.4}$$

Equation (8.4) is the ultimate logic constraint for $[\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft t}$*.*

Example 14. *Consider the tuple type* $\tau = \langle \alpha_1, \mathsf{Int}, [h \mapsto \alpha_2] \rangle$ *and assume that there are at most 4 tuple indices (*$\mathsf{NI} = 4$*) and two record fields (*$\mathsf{NF} = 2$*), while* $\mathsf{fld}(2) = h$*. For a term* t*, the type-term compatibility constraint* $[\![\tau]\!]_{\blacktriangleleft t}$ *is as follows (for fresh variables* $c_1, \ldots, c_6$*):*

$$t = \text{tup}(3, c_1, c_2, c_3, c_4) \wedge [\![\alpha_1]\!]_{\blacktriangleleft c_1} \wedge [\![\mathsf{Int}]\!]_{\blacktriangleleft c_2} \wedge [\![[h \mapsto \alpha_2]]\!]_{\blacktriangleleft c_3}$$

The constraint $[\![\alpha_1]\!]_{\blacktriangleleft c_1}$ *is* $c_1 = \lceil \alpha_1 \rceil$*, while* $[\![\mathsf{Int}]\!]_{\blacktriangleleft c_2}$ *is* $c_2 = \text{int}$*. Finally, the constraint* $[\![\tau_3]\!]_{\blacktriangleleft c_3}$ *is:*

$$c_3 = \text{rec}(c_5, c_6) \wedge c_6 = \lceil \alpha_2 \rceil$$

Note that we know that the tuple term has size 3*, so the variable* c_4 *is not constrained. Similarly,* c_6 *is constrained, but* c_5 *is not (the record type contains only the field with index 2).*

The following proposition shows that our encoding does not introduce inconsistencies:

Proposition 13. *For every type $\tau \in \mathcal{T}^\alpha$ and every fresh variable c, there is a model $\mathcal{M}$ such that $\mathcal{M} \models [\![\tau]\!]_{\blacktriangleleft c}$.*

Proof. This follows trivially; every sub-construction of the shape $[\![\hat{\tau}]\!]_{\blacktriangleleft \hat{c}}$ introduces fresh variables, which means that all generated conjuncts are either a) completely independent, b) they assign a ground term to the fresh variable or c) they assign $\lceil\alpha\rceil$ to the variable, for some α. Since the variables $\lceil\alpha\rceil$ are completely unconstrained, we can easily see that all conjuncts are themselves satisfiable and, since they are either independent or they establish a set of equalities between variables (and, importantly, no ground terms), their conjunction is also satisfiable. □

The following lemma demonstrates, that our definition of type-term compatibility satisfies Property 1, as desired:

Lemma 11. *For every monotype $\tau \in \mathcal{T}$ and every term t, the following holds:*
For a model $\mathcal{M}$ of $[\![\tau]\!]_{\blacktriangleleft t}$, that is $\mathcal{M} \models [\![\tau]\!]_{\blacktriangleleft t}$, it is the case that τ is a subtype of $\tau_{\mathcal{M}}(t)$, that is, $\tau \triangleleft \tau_{\mathcal{M}}(t)$.

Proof. We perform induction on the structure of τ:

τ **is** $\mathsf{Int}, \mathsf{String}$**, or** Bool**:** W.l.o.g., $\tau = \mathsf{Int}$. Since $[\![\tau]\!]_{\blacktriangleleft t}$ is $t = \text{int}$, $\tau_{\mathcal{M}}(\text{int}) = \mathsf{Int}$, and $\triangleleft$ is reflexive, the lemma holds.

$\tau = \mathsf{Set}(\hat{\tau})$**:** Assume, as the induction hypothesis, that the lemma holds for $\hat{\tau}$. By definition, there is a variable c, for which $[\![\tau]\!]_{\blacktriangleleft t}$ is $t = \text{set}(c) \wedge [\![\hat{\tau}]\!]_{\blacktriangleleft c}$ and $\tau_{\mathcal{M}}(t) = \mathsf{Set}(\tau_{\mathcal{M}}(c))$. In particular, $\mathcal{M} \models [\![\hat{\tau}]\!]_{\blacktriangleleft c}$. We can use the induction hypothesis to see that

$$\tau = \mathsf{Set}(\hat{\tau}) \triangleleft \mathsf{Set}(\tau_{\mathcal{M}}(c)) = \tau_{\mathcal{M}}(t)$$

The cases for sequences, functions, and operators are similar.

$\tau = [h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n]$**:** Assume, as the induction hypothesis, that the lemma holds for $\tau_1, \ldots, \tau_n$. Recall that we have a field enumeration fld. Denote by $i(j)$ the fld-preimage of h_j, that is, $\mathsf{fld}(i(j)) = h_j$. By definition, there exist variables $c_1, \ldots, c_{\mathsf{NF}}$, for which $[\![\tau]\!]_{\blacktriangleleft t}$ is

$$t = \text{rec}(c_1, \ldots, c_{\mathsf{NF}}) \wedge \bigwedge_{j=1}^{n} [\![\tau_j]\!]_{\blacktriangleleft c_{i(j)}}$$

and $\tau_{\mathcal{M}}(t)$ is

$$[\mathsf{fld}(1) \mapsto \tau_{\mathcal{M}}(c_1), \ldots, \mathsf{fld}(\mathsf{NF}) \mapsto \tau_{\mathcal{M}}(c_{\mathsf{NF}})]$$

To show that $\tau \triangleleft \tau_{\mathcal{M}}(t)$, the following must hold:

1. $\tau_{\mathcal{M}}(t)$ must have the fields $h_1, \ldots, h_n$, and
2. for every $j \in \{1, \ldots, n\}$, it must be the case that $\tau_j \triangleleft \tau_{\mathcal{M}}(c_{i(j)})$

(1) trivially holds by construction, since $h_j = \mathsf{fld}(i(j))$. To show (2) holds, pick an arbitrary $j \in \{1, \ldots, n\}$. Since $\mathcal{M} \models [\![\tau]\!]_{\blacktriangleleft t}$, that implies $\mathcal{M} \models [\![\tau_j]\!]_{\blacktriangleleft c_{i(j)}}$. We can then use the induction hypothesis for τ_j, to conclude $\tau_j \triangleleft \tau_{\mathcal{M}}(c_{i(j)})$.

The cases for tuples is similar.

$\tau = \langle i_1 \mapsto \tau_1, \ldots, i_n \mapsto \tau_n \rangle$**:** Assume, as the induction hypothesis, that the lemma holds for $\tau_1, \ldots, \tau_n$. By definition, there exist variables $k, c_1, \ldots, c_{\mathsf{NI}}$, for which $[\![\tau]\!]_{\blacktriangleleft t}$ is

$$t = \mathrm{tup}(k, c_1, \ldots, c_{\mathsf{NI}}) \wedge m \leq k \leq \mathsf{NI} \wedge \bigwedge_{j=1}^{n} [\![\tau_j]\!]_{\blacktriangleleft c_{i_j}}$$

where $m = \max\{i_1, \ldots, i_n\}$. To prove that $\tau \triangleleft \tau_{\mathcal{M}}(t)$, it suffices to show the following:

1. $\tau_{\mathcal{M}}(t)$ is a tuple of size at least m.
2. for every $j \in \{1, \ldots, n\}$, it must be the case that $\tau_j \triangleleft \tau_{\mathcal{M}}(c_{i_j})$

(1) follows from the fact that, in particular, $\mathcal{M} \models t = \mathrm{tup}(k, c_1, \ldots, c_{\mathsf{NI}}) \wedge m \leq k \leq \mathsf{NI}$. Let $\hat{k}$ be the $\mathcal{M}$-evaluation of k. We know that $m \leq \hat{k} \leq \mathsf{NI}$ and $\tau_{\mathcal{M}}(t)$ is

$$\langle \tau_{\mathcal{M}}(c_1), \ldots, \tau_{\mathcal{M}}(c_{\hat{k}}) \rangle$$

that is, a tuple of size no less than m. To demonstrate (2), we pick an arbitrary $j \in \{i, \ldots, n\}$. In particular, $\mathcal{M} \models [\![\tau_j]\!]_{\blacktriangleleft c_{i_j}}$, so we can use the induction hypothesis for τ_j, to conclude $\tau_j \triangleleft \tau_{\mathcal{M}}(c_{i_j})$.

□

Observe, however, that type-term compatibility is defined not only for monotypes, but for all types $\tau \in \mathcal{T}^{\alpha}$. The following theorem is a generalization of Lemma 11:

Theorem 5. *For every type $\tau \in \mathcal{T}^{\alpha}$ and every term t, the following holds:*
For a model $\mathcal{M}$ of $[\![\tau]\!]_{\blacktriangleleft t}$, that is $\mathcal{M} \models [\![\tau]\!]_{\blacktriangleleft t}$, consider the substitution σ that assigns to every type variable α the type $\tau_{\mathcal{M}}(\lceil \alpha \rceil)$. Then, $\tau \cdot \sigma$ is a subtype of $\tau_{\mathcal{M}}(t)$, that is, $\tau \cdot \sigma \triangleleft \tau_{\mathcal{M}}(t)$.

The intuition behind the above theorem is that, for a type variable α, using the logic variable $\lceil \alpha \rceil$ results in constraints with many possible models. Picking one such model is equivalent to selecting a concretization of α, i.e. a substitution σ, for which $\alpha \cdot \sigma$ is a monotype. The proof goes along the same lines as the proof of Lemma 11, but we additionally need to pay attention to type substitutions.

Proof. The proof is similar to that of Lemma 11, we perform induction on the structure of τ:

τ is $\mathsf{Int}, \mathsf{String}$, or Bool: W.l.o.g., $\tau = \mathsf{Int}$. By definition, $[\![\tau]\!]_{\blacktriangleleft t}$ is $t = \text{int}$ and $\tau_{\mathcal{M}}(\text{int}) = \mathsf{Int}$. Regardless of σ, $\tau \cdot \sigma = \tau$ and since $\triangleleft$ is reflexive, the lemma holds.

τ is a type variable, α: By construction, $\tau \cdot \sigma = \tau_{\mathcal{M}}(\lceil \alpha \rceil)$. We know that $[\![\tau]\!]_{\blacktriangleleft t}$ is $t = \lceil \alpha \rceil$. Consequently, $\tau_{\mathcal{M}}(t) = \tau_{\mathcal{M}}(\lceil \alpha \rceil)$. As $\triangleleft$ is reflexive, the lemma holds.

$\tau = \mathsf{Set}(\hat{\tau})$: Assume, as the induction hypothesis, that the lemma holds for $\hat{\tau}$. By definition, there is a variable c, for which $[\![\tau]\!]_{\blacktriangleleft t}$ is $t = \text{set}(c) \wedge [\![\hat{\tau}]\!]_{\blacktriangleleft c}$ and $\tau_{\mathcal{M}}(t) = \mathsf{Set}(\tau_{\mathcal{M}}(c))$. Let $\tau_c = \tau_{\mathcal{M}}(c)$. Since $\mathcal{M} \models [\![\hat{\tau}]\!]_{\blacktriangleleft c}$, we can use the induction hypothesis to deduce that $\hat{\tau} \cdot \sigma \triangleleft \tau_c$. By the definition of the subtype relation, $\hat{\tau} \cdot \sigma \triangleleft \tau_c$ implies

$$\tau \cdot \sigma = \mathsf{Set}(\hat{\tau} \cdot \sigma) \triangleleft \mathsf{Set}(\tau_c) = \tau_{\mathcal{M}}(t)$$

The cases for sequences, functions, and operators are similar.

$\tau = [h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n]$: Assume, as the induction hypothesis, that the lemma holds for $\tau_1, \ldots, \tau_n$. Recall that we have a field enumeration fld. Denote by $i(j)$ the fld-preimage of h_j, that is, $\mathsf{fld}(i(j)) = h_j$. By definition, there exist variables $c_1, \ldots, c_{\mathsf{NF}}$, for which $[\![\tau]\!]_{\blacktriangleleft t}$ is

$$t = \text{rec}(c_1, \ldots, c_{\mathsf{NF}}) \wedge \bigwedge_{j=1}^{n} [\![\tau_j]\!]_{\blacktriangleleft c_{i(j)}}$$

and $\tau_{\mathcal{M}}(t)$ is

$$[\mathsf{fld}(1) \mapsto \tau_{\mathcal{M}}(c_1), \ldots, \mathsf{fld}(\mathsf{NF}) \mapsto \tau_{\mathcal{M}}(c_{\mathsf{NF}})]$$

Additionally, the type $\tau \cdot \sigma$ is equal to

$$[h_1 \mapsto \tau_1 \cdot \sigma, \ldots, h_n \mapsto \tau_n \cdot \sigma]$$

To show that $\tau \cdot \sigma \triangleleft \tau_{\mathcal{M}}(t)$, the following must hold:

1. $\tau_{\mathcal{M}}(t)$ must have the fields $h_1, \ldots, h_n$, and
2. for every $j \in \{1, \ldots, n\}$, it must be the case that $\tau_j \cdot \sigma \triangleleft \tau_{\mathcal{M}}(c_{i(j)})$

(1) trivially holds by construction, since $h_j = \mathsf{fld}(i(j))$. To show (2) holds, pick any $j \in \{1, \ldots, n\}$. Since $\mathcal{M} \models [\![\tau]\!]_{\blacktriangleleft t}$, that implies $\mathcal{M} \models [\![\tau_j]\!]_{\blacktriangleleft c_{i(j)}}$. We can then use the induction hypothesis for τ_j, to conclude $\tau_j \cdot \sigma \triangleleft \tau_{\mathcal{M}}(c_{i(j)})$.

The cases for tuples and sparse tuples are similar.

□

8.7 Logical encoding of type schemas

In this section, we show how to encode schema instances, i.e. create constraints specifying that the arguments and result of an operator application (or rather, the FOL variables that represent them) hold the types that make up a schema (up to subtyping).

8.7.1 Instances of primitive type schemas

In this section, we focus on primitive schemas of built-in operators. A primitive schema s for an operator F of arity n has the shape:

$$\forall \alpha_1, \ldots, \alpha_k \,.\, \langle \omega_1, \cdots \omega_n \rangle \Rightarrow \hat{\omega},$$

where types $\omega_1, \ldots, \omega_n, \hat{\omega}$ may contain type variables $\alpha_1, \ldots, \alpha_k$ in any combination.

An application of operator F gives us $n+1$ constraints: n constraints for the operator arguments and one constraint for the result. Assume that terms $t_1, \ldots, t_n$ encode the types of the arguments and a term $\hat{t}$ encodes the type of the result. Then, we define a *schema instance* $[\![s]\!]_{\langle t_1,\ldots,t_n|\hat{t}\rangle}$. It is a logic constraint that instantiates schema s with the terms $t_1, \ldots, t_n, \hat{t}$ as follows:

1. Introduce fresh type variables $\beta_1, \ldots, \beta_k$, which represent the (local) instantiations of the type parameters $\alpha_1, \ldots, \alpha_k$.
2. Construct the type substitution σ that maps $\alpha_1, \ldots, \alpha_k$ to $\beta_1, \ldots, \beta_k$, respectively.
3. Construct types $\chi_1, \ldots, \chi_n, \hat{\chi}$ by applying the substitution σ to the types $\omega_1, \ldots, \omega_n, \hat{\omega}$, respectively.
4. Finally, define the instance $[\![s]\!]_{\langle t_1,\ldots,t_n|\hat{t}\rangle}$ as $[\![\chi_1]\!]_{\blacktriangleleft t_1} \wedge \cdots \wedge [\![\chi_n]\!]_{\blacktriangleleft t_n} \wedge [\![\hat{\chi}]\!]_{\blacktriangleleft \hat{t}}$.

Example 15. *Consider the set union $\cup$, which has the schema*

$$s_\cup : \forall \alpha \,.\, \langle \mathsf{Set}(\alpha), \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Set}(\alpha)$$

In Example 13, we showed that $[\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft t}$ equals (for a fresh c and $b = \lceil \beta \rceil$):

$$t = \mathrm{set}\,(c) \wedge c = b$$

To define a schema instance $[\![s_\cup]\!]_{\langle t_1,t_2|\hat{t}\rangle}$, we apply the above expression 3 times (for $t_1, t_2, \hat{t}$):

$$[\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft t_1} \wedge [\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft t_2} \wedge [\![\mathsf{Set}(\beta)]\!]_{\blacktriangleleft \hat{t}}$$

As a result, we obtain the constraint for the schema instance:

$$t_1 = \mathrm{set}(c_1) \wedge c_1 = b \wedge t_2 = \mathrm{set}(c_2) \wedge c_2 = b \wedge \hat{t} = \mathrm{set}\,(\hat{c}) \wedge \hat{c} = b$$

This matches our intuition that both the arguments and the result of a set union are sets that contain elements of the same type.

The only TLA$^+$ operator that needs special treatment is the EXCEPT operator that partially updates a function, e.g., $e_4 = [e_1 \text{ EXCEPT } ![e_2] = e_3]$. We have to add an additional constraint that the result type is equal to the type of the first argument (the function under modification). A bit more formally, we would add the constraint: $\hat{t} = t_1$.

In the following lemma, we show the connection between schema instances and the rule (BUILTIN) for the built-in operators in Section 8.5:

Lemma 12. *Let F be an operator that is annotated with a primitive schema s, that is, s has the form $\forall\alpha_1, \ldots \alpha_k \,.\, \langle\omega_1, \ldots, \omega_n\rangle \Rightarrow \hat{\omega}$ and let Γ be an arbitrary type environment. If for a model $\mathcal{M}$ and terms $t_1, \ldots, t_n, \hat{t}$, it holds that $\mathcal{M} \models [\![s]\!]_{\langle t_1,\ldots,t_n|\hat{t}\rangle}$, then the following holds:*

$$\Gamma \vdash F\colon \langle\tau_{\mathcal{M}}(t_1), \ldots, \tau_{\mathcal{M}}(t_n)\rangle \Rightarrow \tau_{\mathcal{M}}(\hat{t})$$

Proof. By definition, there exist type variables $\beta_1, \ldots, \beta_k$ and a substitution σ, replacing all occurrences of $\alpha_1, \ldots, \alpha_k$ with $\beta_1, \ldots, \beta_k$. Denote with $\chi_1, \ldots, \chi_n, \hat{\chi}$ the types $\omega_1 \cdot \sigma, \ldots, \omega_n \cdot \sigma, \hat{\omega} \cdot \sigma$. Then, $[\![s]\!]_{\langle t_1,\ldots,t_n|\hat{t}\rangle}$ is

$$[\![\hat{\chi}]\!]_{\blacktriangleleft\hat{t}} \wedge \bigwedge_{i=1}^{n} [\![\chi_i]\!]_{\blacktriangleleft t_i}$$

We can use Theorem 5 on $\chi_1, \ldots, \chi_n, \hat{\chi}$, to assert that $\hat{\chi} \cdot \sigma_\beta \triangleleft \tau_{\mathcal{M}}(\hat{t})$ and $\chi_i \cdot \sigma_\beta \triangleleft \tau_{\mathcal{M}}(t_i)$ for all $i \in \{1, \ldots, n\}$, where σ_β is the substitution replacing each β_j with $\tau_{\mathcal{M}}(\lceil\beta_j\rceil)$ for $j \in \{1, \ldots, k\}$.

To use the derivation rule (BUILTINOP), we first make the following observation: because of the nature of the construction of χ_i from ω_i (resp. $\hat{\chi}$ from $\hat{\omega}$), the types

$$(\langle\omega_1, \ldots, \omega_n\rangle \Rightarrow \hat{\omega})[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]$$

and

$$(\langle\chi_1, \ldots, \chi_n\rangle \Rightarrow \hat{\chi})[\tau_1/\beta_1, \ldots, \tau_k/\beta_k]$$

are equal for all selections of $\tau_1, \ldots, \tau_k$, since

$$[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k] = [\tau_1/\beta_1, \ldots, \tau_k/\beta_k] \circ \sigma$$

Let us pick $\tau_1 = \tau_{\mathcal{M}}(\lceil\beta_1\rceil), \ldots, \tau_k = \tau_{\mathcal{M}}(\lceil\beta_k\rceil)$ and denote by σ_α the substitution $[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k]$, i.e., the substitution $[\tau_{\mathcal{M}}(\lceil\beta_1\rceil)/\alpha_1, \ldots, \tau_{\mathcal{M}}(\lceil\beta_k\rceil)/\alpha_k]$

By (BUILTINOP), we can derive the following:

$$\Gamma \vdash F\colon (\langle\omega_1, \ldots, \omega_n\rangle \Rightarrow \hat{\omega}) \cdot \sigma_\alpha$$

or equivalently

$$\Gamma \vdash F\colon \langle \omega_1 \cdot \sigma_\alpha, \ldots, \omega_n \cdot \sigma_\alpha \rangle \Rightarrow \hat{\omega} \cdot \sigma_\alpha$$

If we can demonstrate that $\hat{\omega} \cdot \sigma_\alpha \triangleleft \tau_{\mathcal{M}}(\hat{t})$ and $\omega_i \cdot \sigma_\alpha \triangleleft \tau_{\mathcal{M}}(t_i)$ for all $i \in \{1, \ldots, n\}$, we can use the (Sub) rule to assert that

$$\Gamma \vdash F\colon \langle \tau_{\mathcal{M}}(t_1), \ldots, \tau_{\mathcal{M}}(t_n) \rangle \Rightarrow \tau_{\mathcal{M}}(\hat{t})$$

To that end, take an arbitrary $j \in \{1, \ldots, n\}$. We know that $\chi_i \cdot \sigma_\beta \triangleleft \tau_{\mathcal{M}}(t_i)$. It remains to be seen that $\chi_i \cdot \sigma_\beta = \omega_i \cdot \sigma_\alpha$. By definition, $\chi_i = \omega_i \cdot \sigma$, so the problem is reduced to showing that $\sigma_\alpha = \sigma_\beta \circ \sigma$. But this follows from the above observation that

$$[\tau_1/\alpha_1, \ldots, \tau_k/\alpha_k] = [\tau_1/\beta_1, \ldots, \tau_k/\beta_k] \circ \sigma$$

for all selections of $\tau_1, \ldots, \tau_k$, as σ_β is exactly $[\tau_1/\beta_1, \ldots, \tau_k/\beta_k]$ for the particular selection of $\tau_1 = \tau_{\mathcal{M}}(\lceil \beta_1 \rceil), \ldots, \tau_k = \tau_{\mathcal{M}}(\lceil \beta_k \rceil)$. The proof of $\hat{\omega} \cdot \sigma_\alpha \triangleleft \tau_{\mathcal{M}}(\hat{t})$ is identical. Therefore, we know the subtype relations hold as desired and the lemma holds. □

8.7.2 Instances of complex type schemas (Overloaded Operators)

As we have mentioned, operators may be overloaded and consequently may be associated with complex schemas. Consider a complex schema s of the shape $s_1 \sqcup \ldots \sqcup s_m$. We define an instance of s for terms $t_1, \ldots, t_n, \hat{t}$ as the disjunction of the instances constructed for the primitive schemas $s_1, \ldots, s_m$. Formally, $[\![s_1 \sqcup \ldots \sqcup s_m]\!]_{\langle t_1, \ldots, t_n | \hat{t} \rangle}$ is defined as:

$$[\![s_1]\!]_{\langle t_1, \ldots, t_n | \hat{t} \rangle} \vee \cdots \vee [\![s_m]\!]_{\langle t_1, \ldots, t_n | \hat{t} \rangle}$$

Example 16. *Consider the TLA$^+$ expression $f[2]$, in which f can be either a function, a tuple, or a sequence. This instance of $\_[\_]$ has the following complex schema $s_1 \sqcup s_2 \sqcup s_3$:*

$$\forall \alpha_1, \alpha_2 \,.\, \langle \alpha_1 \to \alpha_2, \alpha_1 \rangle \Rightarrow \alpha_2 \sqcup \forall \alpha_3 \,.\, \langle \langle 2 \mapsto \alpha_3 \rangle, \mathsf{Int} \rangle \Rightarrow \alpha_3 \sqcup \forall \alpha_4 \,.\, \langle \mathsf{Seq}(\alpha_4), \mathsf{Int} \rangle \Rightarrow \alpha_4$$

Schema s_1 corresponds to function application, schema s_2 corresponds to tuple access, and schema s_3 corresponds to sequence access. Importantly, schema s_2 captures specifically access to the second tuple element. (If the index expression was non-constant, we would not consider tuple access among the available schema options.) Moreover, schema s_2 uses the sparse tuple type $\langle 2 \mapsto \alpha_3 \rangle$, as otherwise we would have to introduce one schema per tuple size, which is unknown at this point.

Assume that the number of tuple indices is bounded by 3, that is, $\mathsf{NI} = 3$. Then, the instance $[\![s_1 \sqcup s_2 \sqcup s_3]\!]_{\langle t_1, t_2 | \hat{t} \rangle}$ of the schema $s_1 \sqcup s_2 \sqcup s_3$ applied to terms $t_1, t_2, \hat{t}$ is given below (the type variables $\alpha_1, \ldots, \alpha_4$ are instantiated with $\beta_1, \ldots, \beta_4$, and $c_1, \ldots, c_7$ are fresh constants):

$$\begin{aligned}
&\Big(\hat{t} = \lceil \beta_2 \rceil \wedge t_1 = \mathrm{fun}\,(c_1, c_2) \wedge c_1 = \lceil \beta_1 \rceil \wedge c_2 = \lceil \beta_2 \rceil \wedge t_2 = \lceil \beta_1 \rceil\Big) \\
\vee &\Big(\hat{t} = \lceil \beta_3 \rceil \wedge t_1 = \mathrm{tup}(c_3, c_4, c_5, c_6) \wedge c_5 = \lceil \beta_3 \rceil \wedge t_2 = \mathrm{int} \wedge 2 \leq c_3 \leq 3\Big) \\
\vee &\Big(\hat{t} = \lceil \beta_4 \rceil \wedge t_1 = \mathrm{seq}\,(c_7) \wedge c_7 = \lceil \beta_4 \rceil) \wedge t_2 = \mathrm{int}\Big)
\end{aligned}$$

As one can see from this example, complex type schemas give rise to disjunctive constraints, whereas all other logic constructions are conjunctive.

Similar to Lemma 12, we prove the connection between schema instances and the rule (BuiltIn) for the complex schemas:

Lemma 13. *Let F be an operator that is annotated with a complex schema s of the form $s_1 \sqcup \ldots \sqcup s_m$ and let Γ be an arbitrary type environment. If for a model $\mathcal{M}$ and terms $t_1, \ldots, t_n, \hat{t}$, it holds that $\mathcal{M} \models [\![s]\!]_{\langle t_1,\ldots,t_n|\hat{t}\rangle}$, then the following holds:*

$$\Gamma \vdash F\colon \langle \tau_{\mathcal{M}}(t_1), \ldots, \tau_{\mathcal{M}}(t_n)\rangle \Rightarrow \tau_{\mathcal{M}}(\hat{t})$$

Proof. Follows from the fact that the schema instance for a complex schema $s_1 \sqcup \ldots \sqcup s_m$ is the disjunction of schema instances corresponding to the primitive schemas $s_1, \ldots, s_m$ and the fact that the (BuiltInOp) rule allows us to choose which schema component to use in the derivation. □

8.8 Logic constraints for type inference

Having defined a logic encoding for types in Section 8.6 and schema instances in Section 8.7, we are now in a position to translate the type inference problem in first-order logic for arbitrary expressions. In our approach, determining type constraints for a TLA$^+$ expression amounts to producing constraints by instantiating schemas of the built-in operators. However, we have one more language feature to address, namely, user-defined operators. Instead of seeking the most-general types of user-defined operators (principal types), we simply infer the types for all occurrences of user-defined operators either at their application sites or their use as arguments to higher-order operators. By doing so, we avoid quantification at the logic level (and subsequently in SMT).

8.8.1 Definition cloning at call-sites

Before we delve into generating constraints in the general case, we further preprocess the specification. Since we do not compute anything similar to principal types for user-defined operators, we can clone the definitions of these operators at the call sites. This is similar to instantiation of schemas. In contrast to [DM82], we are not generalizing the types from these different instances, and thus we are computing a finite number of monotypes for every user-defined operator.

Consider a user-defined operator F with the definition $F(p_1, \ldots, p_n) \triangleq e_F$. The operator F can be applied to different arguments that would require us to assign multiple monotypes to F. Instead of collecting monotypes for F, we simply clone F at every application site. Whenever F is applied to the arguments $e_1, \ldots, e_n$, we introduce fresh identifiers for the operator name and its parameters $\hat{F}, \hat{p}_1, \ldots, \hat{p}_n$ and define $e_{\hat{F}}$ as the

```
1  \* Before cloning:
2  LET mem(e, f) ≜
3    ∃ i ∈ DOMAIN f: e = f[i]
4  IN mem(1, ⟨2, 3⟩)
5    ∧ mem("a", [g ∈ {1, 2} ↦ "b"])
```

```
1  \* After cloning:
2  LET mem1(e1, f1) ≜
3    ∃ i1 ∈ DOMAIN f1: e1 = f1[i1]
4  IN mem1(1, ⟨2, 3⟩)
5   ∧
6  LET mem2(e2, f2) ≜
7    ∃ i2 ∈ DOMAIN f2: e2 = f2[i2]
8  IN mem2("a", [g ∈ {1, 2} ↦ "b"])
```

Figure 8.4: An example of cloning the definitions at operator application sites

TLA$^+$ expression obtained by replacing every occurrence of p_i in e with $\hat{p}_i$, and every occurrence of F with $\hat{F}$. Then we replace $F(e_1, \ldots, e_n)$ with the LET-IN definition:

$$\text{LET } \hat{F}(\hat{p}_1, \ldots, \hat{p}_n) \triangleq e_{\hat{F}} \text{ IN } \hat{F}(e_1, \ldots, e_k)$$

Remark 4. *If the user-defined operators are recursion-free, then the above transformation is simple. In case of recursion, or mutual recursion, we have to first find the cycles in the call graph and simultaneously clone the involved recursive operators. We do not give a formal definition here, as it does not bring new insights. Moreover, mutually-recursive operators are quite rare in TLA$^+$.*

Another case, where the operator F may be assigned multiple monotypes, is when F is passed as an argument to a higher-order operator. (However, recall that a higher-order operator cannot be passed to another higher-order operator in TLA$^+$.) Assume that F is passed to a higher-order operator G as $G(\ldots, F, \ldots)$. We produce a fresh copy $\hat{F}$ of F as in the above case and replace $G(\ldots, F, \ldots)$ with the LET-IN definition:

$$\text{LET } \hat{F}(\hat{p}_1, \ldots, \hat{p}_n) \triangleq \hat{e} \text{ IN } G(\ldots, \hat{F}, \ldots)$$

Example 17. *Figure 8.4 shows a simple example of definition cloning.*

Remark 5. *In the implementation, we do not clone operator definitions, but produce the constraints dynamically at every application site. By collecting the call stack, we also correctly deal with the recursive operators. However, we find that this technical detail obfuscates the core idea. Hence, we chose to explain the technique by performing the above transformation.*

8.8.2 Producing first-order type constraints for TLA$^+$ expressions

In a similar fashion to how we handled type variables in type-term compatibility constraints, we introduce a global mapping Δ, that assigns to a name $x \in \mathsf{Names}$ a fixed first-order term. Similar to a type environment Γ, this mapping assigns terms to the

names, which can be used to reconstruct types from a first-order interpretation. Therefore, we call Δ a *term environment*. In contrast to Γ, we make Δ static by pre-populating it with the terms for all names:

- When x is the name of either a user-defined operator, or operator parameter of arity n, we pick fresh logic variables $\hat{c}, c_1, \ldots, c_n$ and define $\Delta(x) = \text{oper}_n(c_1, \ldots, c_n, \hat{c})$
- When x is a specification variable, we pick a fresh logic variable c and define $\Delta(x) = c$ and $\Delta(x') = c$ (to ensure the types of specification variables do not change during the execution).
- When x is a specification constant, simple operator parameter, or bound variable, we pick a fresh logic variable c and define $\Delta(x) = c$.

To produce constraints for TLA$^+$ expressions in general, we define a translation $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ that receives two arguments: a term t, and a TLA$^+$ expression e. Informally, $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ defines the constraints, which capture that e may be assigned the type represented by the term t. We define $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ recursively on the structure of e:

- We define $[\![i]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ as $t = \text{int}$ for an integer literal i. Similar for strings and Booleans.
- We define $[\![x]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ as $t = \Delta(x)$ for $x \in \mathsf{Names}$.
- If e is an application $F(e_1, \ldots, e_n)$ of a built-in operator F, then F is annotated with a schema s. We introduce fresh variables $c_1, \ldots, c_n$ and use a schema instance to define $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$:

$$[\![s]\!]_{\langle c_1, \ldots, c_n | t \rangle} \wedge [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![e_n]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_n}$$

- If e is an application $F(e_1, \ldots, e_n)$ of a user-defined operator F, then $\Delta(F)$ has the shape $\text{oper}_n(c_1, \ldots, c_n, \hat{c})$. We use this to define $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$:

$$t = \hat{c} \wedge [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![e_n]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_n}$$

- If e is a LET-IN expression LET $F(p_1, \ldots, p_n) \triangleq e_1$ IN e_2, then $\Delta(F)$ has the operator shape $\text{oper}_n(c_1, \ldots, c_n, \hat{c})$. We define $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ as:

$$[\![p_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![p_n]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_n} \wedge [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft \hat{c}} \wedge [\![e_2]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$$

  The case of recursive let-in expressions is identical.

Remark 6. *One can notice that the above translation does not explicitly include rules similar to (Exists) and (Map) that are present in Figure 8.2 of Section 8.5. The reasons for that are two-fold:*

1. *The rules (Exists) and (Map) in Section 8.5 introduce new names in the type environment, whereas we pre-populate the term context before doing the translation* $[\![e]\!]^{TLA}_{\blacktriangleleft t}$.

2. *The constructs such as* $\exists x \in S : P$ *and* $\{e : x \in S\}$ *are considered as operator applications in TLA*$^+$. *Hence, we have introduced schemas for them, and these operators are handled as the other built-in operators.*

These observations allow us to minimize the number of cases to consider in the translation.

8.8.3 Main theorem

Suppose we are given an interpretation $\mathcal{M}$. The term environment Δ naturally defines a type environment $\Gamma^{\Delta}_{\mathcal{M}}$ in the following way: for every x in the domain of Δ, we define $\Gamma^{\Delta}_{\mathcal{M}}(x)$ to be $\tau_{\mathcal{M}}(\Delta(x))$.

We can then ask the following question: *If we take a TLA*$^+$ *expression* e *and a model* $\mathcal{M}$ *of* $[\![e]\!]^{TLA}_{\blacktriangleleft t}$, *what is the relation between* $\tau_{\mathcal{M}}(t)$ *and* $\Gamma^{\Delta}_{\mathcal{M}}(e)$*?*

We state the main result of this chapter:

Theorem 6 (Soundness)**.** *Fix a TLA*$^+$ *specification. For every TLA*$^+$ *expression* e *in the specification and every term* t *of the sort* S_T*, the following holds:*
If $\mathcal{M}$ *is a model of* $[\![e]\!]^{TLA}_{\blacktriangleleft t}$, *that is,* $\mathcal{M} \models [\![e]\!]^{TLA}_{\blacktriangleleft t}$, *and* $\Gamma^{\Delta}_{\mathcal{M}}$ *is the type environment defined by* Δ *and* $\mathcal{M}$*, the type constructed from the term* t *can be inferred for the expression* e *in the type environment* $\Gamma^{\Delta}_{\mathcal{M}}$, *that is,* $\Gamma^{\Delta}_{\mathcal{M}} \vdash e : \tau_{\mathcal{M}}(t)$.

Note that Theorem 6 holds for all expressions e, not just identifiers in the domain of Δ, which tells us that we can trust the types constructed by $\tau_{\mathcal{M}}$, as they are permissible within the type derivation framework.

Proof. We perform induction on the structure of e.

e **is a literal** l**:** Assume, w.l.o.g., that l is an integer literal. By definition, $[\![l]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ is $t = \text{int}$. Since $\mathcal{M} \models t = \text{int}$, clearly $\tau_{\mathcal{M}}(t) = \mathsf{Int}$. As l is an integer literal, we can use the (Int) rule to derive

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash l : \mathsf{Int}$$

e **is an identifier** $x \in$ **Names:** By definition, $[\![x]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ is $t = \Delta(x)$. We can use the (Env) rule to derive

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash x : \Gamma^{\Delta}_{\mathcal{M}}(x)$$

By definition, $\Gamma^{\Delta}_{\mathcal{M}}(x) = \tau_{\mathcal{M}}(\Delta(x))$, so the theorem holds.

e is an application $F(e_1, \ldots, e_n)$ of a built-in operator F: Assume, as the induction hypothesis, that the theorem holds for $e_1, \ldots, e_n$.

We know that F is associated with a schema s and there exist variables $c_1, \ldots, c_n$, for which $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ is defied as

$$[\![s]\!]_{\langle c_1,\ldots,c_n|t\rangle} \wedge [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![e_n]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_n}$$

We can use the (App) rule to derive $\Gamma^{\Delta}_{\mathcal{M}} \vdash e\colon \tau_{\mathcal{M}}(t)$, if we can demonstrate the following:

1. $\Gamma^{\Delta}_{\mathcal{M}} \vdash F\colon \langle \tau_{\mathcal{M}}(c_1), \ldots, \tau_{\mathcal{M}}(c_n)\rangle \Rightarrow \tau_{\mathcal{M}}(t)$, and
2. $\Gamma^{\Delta}_{\mathcal{M}} \vdash e_i\colon \tau_{\mathcal{M}}(c_i)$ for $i \in \{1, \ldots, n\}$

To see that (1) holds, consider the following: in particular, $\mathcal{M} \models [\![s]\!]_{\langle c_1,\ldots,c_n|t\rangle}$, so we can use Lemma 11 or Lemma 13 to conclude

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash F\colon \langle \tau_{\mathcal{M}}(c_1), \ldots, \tau_{\mathcal{M}}(c_n)\rangle \Rightarrow \tau_{\mathcal{M}}(t)$$

To see that (2) holds, pick an arbitrary $i \in \{1, \ldots, n\}$. In particular, $\mathcal{M} \models [\![e_i]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_i}$, so by the induction hypothesis

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash e_i\colon \tau_{\mathcal{M}}(c_i)$$

Since both conditions hold we can use the (App) rule and the theorem holds.

e is an application $F(e_1, \ldots, e_n)$ of a user-defined operator F: Assume, as the induction hypothesis, that the theorem holds for $e_1, \ldots, e_n$.

By construction, $\Delta(F)$ has the shape $\mathrm{oper}_n(c_1, \ldots, c_n, \hat{c})$ and $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ is defined as:

$$t = \hat{c} \wedge [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![e_n]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_n}$$

Using the (Env) rule, we can assert that

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash F\colon \Gamma^{\Delta}_{\mathcal{M}}(F)$$

Since $\Gamma^{\Delta}_{\mathcal{M}}(F) = \tau_{\mathcal{M}}(\Delta(F))$ and $\tau_{\mathcal{M}}(\mathrm{oper}_n(c_1, \ldots, c_n, \hat{c}))$ is

$$\langle \tau_{\mathcal{M}}(c_1), \ldots, \tau_{\mathcal{M}}(c_n)\rangle \Rightarrow \tau_{\mathcal{M}}(\hat{c})$$

we can conclude

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash F\colon \langle \tau_{\mathcal{M}}(c_1), \ldots, \tau_{\mathcal{M}}(c_n)\rangle \Rightarrow \tau_{\mathcal{M}}(\hat{c})$$

For the same reason as in the case of built in operators, we can see that $\Gamma^{\Delta}_{\mathcal{M}} \vdash e_i\colon \tau_{\mathcal{M}}(c_i)$ for $i \in \{1, \ldots, n\}$, so applying the (App) rule allows us to conclude

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash e\colon \tau_{\mathcal{M}}(\hat{c})$$

Finally, we use the fact that $\mathcal{M} \models t = \hat{c}$, which naturally implies $\tau_{\mathcal{M}}(t) = \tau_{\mathcal{M}}(\hat{c})$, to show that $\Gamma^{\Delta}_{\mathcal{M}} \vdash e\colon \tau_{\mathcal{M}}(t)$ and the theorem holds.

e **is a let-in expression let** $F(p_1, \ldots, p_n) \triangleq e_1$ **in** e_2**:** Assume, as the induction hypothesis, that the theorem holds for e_1, e_2.

By construction, $\Delta(F)$ has the shape $\text{oper}_n(c_1, \ldots, c_n, \hat{c})$ and $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ is defined as:

$$[\![p_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_1} \wedge \cdots \wedge [\![p_n]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_n} \wedge [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft \hat{c}} \wedge [\![e_2]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$$

We will demonstrate that we can use the (Let) rule, for which the following must hold: There exist types $\rho_1, \ldots, \rho_n, \tau_1$, such that

1. We can derive
$$\Gamma^{\Delta}_{\mathcal{M}}, p_1 : \rho_1, \ldots, p_n : \rho_n \vdash e_1 : \tau_1$$
2. We can derive
$$\Gamma^{\Delta}_{\mathcal{M}}, F : \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \tau_1 \vdash e_2 : \tau_{\mathcal{M}}(t)$$

Fix $\rho_1 = \tau_{\mathcal{M}}(c_1), \ldots, \rho_n = \tau_{\mathcal{M}}(c_n)$ and $\tau_1 = \tau_{\mathcal{M}}(\hat{c})$. For these selections, it is actually the case that both $\Gamma^{\Delta}_{\mathcal{M}}, p_1 : \rho_1, \ldots, p_n : \rho_n$ and $\Gamma^{\Delta}_{\mathcal{M}}, F : \langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \tau_1$ are the same as $\Gamma^{\Delta}_{\mathcal{M}}$. The reason for this is as follows: since $\mathcal{M} \models [\![p_i]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_i}$ for all $i \in \{1, \ldots, n\}$ and $[\![p_i]\!]^{\mathsf{TLA}}_{\blacktriangleleft c_i}$ equals $c_i = \Delta(p_i)$ it must be the case that $\Gamma^{\Delta}_{\mathcal{M}}(p_i) = \tau_{\mathcal{M}}(\Delta(p_i)) = \tau_{\mathcal{M}}(c_i)$. Similarly, $\Gamma^{\Delta}_{\mathcal{M}}(F) = \tau_{\mathcal{M}}(\Delta(F)) = \langle \tau_{\mathcal{M}}(c_1), \ldots, \tau_{\mathcal{M}}(c_n) \rangle \Rightarrow \tau_{\mathcal{M}}(\hat{c})$.

Because, in particular, $\mathcal{M} \models [\![e_1]\!]^{\mathsf{TLA}}_{\blacktriangleleft \hat{c}}$, we can use the induction hypothesis to conclude that

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash e_1 : \tau_{\mathcal{M}}(\hat{c})$$

which means (1) holds. Similarly, $\mathcal{M} \models [\![e_2]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ implies

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash e_2 : \tau_{\mathcal{M}}(t)$$

which means (2) holds as well. We can then derive

$$\Gamma^{\Delta}_{\mathcal{M}} \vdash e : \tau_{\mathcal{M}}(t)$$

using the (Let) rule and the theorem holds.

The case of recursive let-in expressions is identical.

□

8.9 Implementation and experiments

We have implemented our approach to type inference on top of the infrastructure provided by TLA Toolbox [KLR19b] and Apalache [KKT19b]. Our tool is implemented in Scala. Our implementation encodes the type inference query as an SMT assertion in Microsoft Z3 [DB08], in the theory of quantifier-free uninterpreted functions. (For convenience, our implementation also uses bounded integers to reason about tuple sizes, which is not required by the theoretical framework in this chapter.)

Figure 8.5: Benchmark sizes

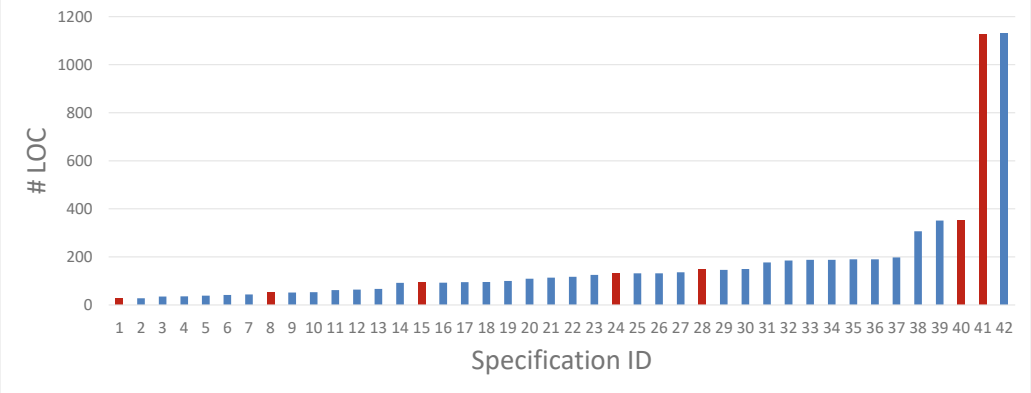
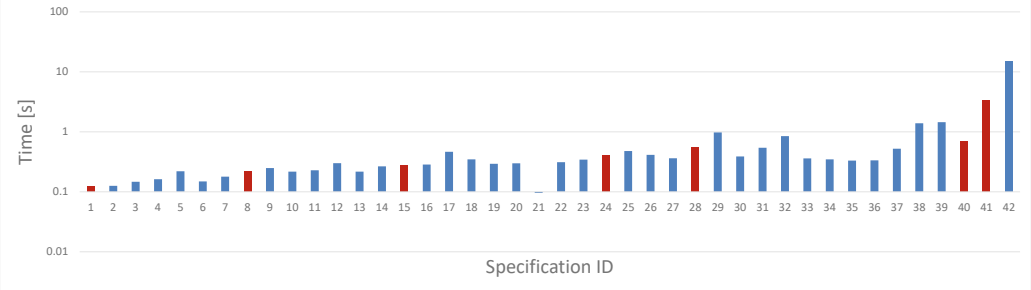


Figure 8.6: Type inference times

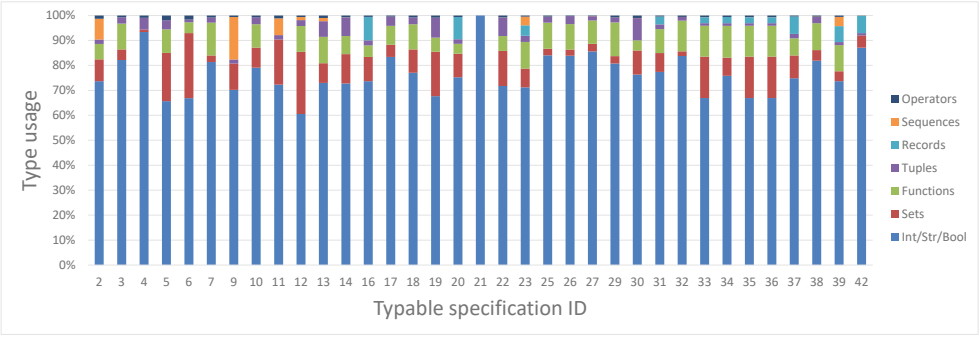


8.9.1 Benchmarks

Our specifications were sampled from the public repository of TLA$^+$ benchmarks maintained by the developers of TLA$^+$ Toolbox [TLA20]. They range from logical puzzles to concurrent algorithms and fault-tolerant distributed algorithms and include examples like Lamport's Paxos (`Paxos.tla`), Dijkstra's Termination Detection (`EWD840.tla`), and the Chang-Roberts leader election algorithm (`ChangRoberts.tla`). Figure 8.5 shows the full list of benchmarks considered, sorted by their size (in terms of LOC). We did not do any preprocessing of the benchmarks to run our implementation. As we wrote our type-inference tool on top of the Apalache infrastructure, we had to exclude some benchmarks that were not completely parsed by the Apalache parser. We have highlighted in red those specifications that contained type errors and have introduced minimal corrections to make them typable - the specifications suffixed "-Fix" are the results of these corrections. The mapping from specification IDs to specifications can be found in Table 8.8.

For each benchmark, Figure 8.6 shows the runtime (in seconds). We were also interested to see what kinds of types the specifications were using, so Figure 8.7 shows the percentage breakdown of type usage in the type-correct specifications.

Figure 8.7: Type usage breakdown



8.9.2 Evaluation

Due to the low computational intensity, all experiments were performed on a Lenovo Thinkpad T470s laptop (Intel Core i7-7500U CPU 2.70GHz x4, 16 GB RAM) and measure wall-time. We have found three kinds of type-incorrect specifications.

The first kind of type errors, and simplest, are those that contain typos, the most common example being mistaking the syntax for a single function and a set of functions in initializers:

$$pc' \in [i \in Proc \mapsto \text{“V0”}] \tag{8.5}$$

In the example in Equation 8.5, the specification designer probably wanted to assign a function to the variable *pc* by picking it from a set of functions. However, the right-hand side of the expression defines a single function. The issue can be fixed by either replacing the first set membership with equality, or by introducing the set of functions, which has similar syntax:

$$pc' \in [Proc \to \{\text{“V0”}\}] \tag{8.6}$$

The second kind was specifications where the authors make no distinction between accessible objects of different types. To clarify, according to the design philosophy of TLA$^+$, the objects $\langle 1, 2\rangle$ and $[i \in \{1, 2\} \mapsto i]$ are indistinguishable; both are (mathematical) functions, with the same domain and evaluating either (by using the $[\cdot]$ operator) yields the same results for all elements of their shared domain $\{1, 2\}$. Thus, for example in the specification `Queens.tla`, a snippet of which can be found in Figure 8.8, the sequence operator *Len* was being called on a function, which we consider type-incorrect.

The third kind, of which the only example in this set of benchmarks was the specification of Dijkstra's mutex (`DijkstraMutex.tla`), a snippet of which can be found in Figure

Figure 8.8: A snippet from `Queens.tla`: *queens* is treated as both a function and a sequence.

MODULE *Queens*

```
...
IsSolution(queens) ≜ ∀i ∈ 1 .. (Len(queens) − 1): ...
Solutions ≜ {queens ∈ [1 .. N → 1 .. N]: IsSolution(queens)}
...
```

Figure 8.9: A snippet from DijkstraMutex.tla: $temp[self]$ is treated as both an integer and a set of integers.

MODULE *DijkstraMutex*

```
...
Init ≜ k ∈ Proc ∧ ...
Li3a(self) ≜ temp′ = [temp EXCEPT ![self] = k] ∧ ...
Li4a(self) ≜ temp′ = [temp EXCEPT ![self] = Proc \ {self}] ∧ ...
...
```

8.9, are specifications that would be dynamically typable, but not statically typable. Concretely, the aforementioned specification uses a variable named *temp* to store values between logical phases of the algorithm. Its range sometimes contains a single process ID (that is, String) and sometimes a set of process IDs (that is, Set(String)). The specification relies on the fact that the control flow is strictly enforced by the operator preconditions for correctness which includes what could be considered dynamic type-correctness. Indeed, if the specification-induced control flow is respected, the variable *temp* is handled correctly, with respect to the type its currently expected to have (i.e. Int or Set(Int)). This example perfectly illustrates the challenges of designing a type-checker for TLA$^+$; because specifications are nothing more than a collection of operators — equivalently, a collection of logical formulas — notions such as control flow cannot be used in type-checking because, unlike programming languages, syntax rarely dictates execution order.

Our implementation is, to our knowledge, the first type inference tool for the whole of TLA$^+$, not just the operator-free fragment or proof obligations. Thus, there is no baseline tool to compare the performance. Although our implementation is a prototype and has not been heavily optimized, the type inference runtime rarely exceed 1 second on our benchmarks. The benchmarks are not significantly smaller than the practical TLA$^+$ specifications, which, unlike source code of real implementations, rarely exceeds a few thousand lines of code. Therefore, we are confident that this approach is usable in practice and could be integrated in tools such as the TLA$^+$ Toolbox and the Apalache model checker. Additionally, all type-checks performed on our benchmarks, and thus all the SMT encodings, were monolithic. In practice, specifications are written incrementally

and the user would often need only check local type-correctness of a single operator to find bugs in their implementation, which would be considerably faster.

8.10 Related work

To our knowledge, the first type system for TLA$^+$ was designed by Merz and Vanzetto [MV12a]. They introduce a basic type system and type inference rules for a large number of operators in a special fragment of TLA$^+$. In contrast, our framework is conceptually simpler, due to offloading the work to the annotations of the built-in operators. The main goal of their work is to translate proof obligations into SMT. As this work extends the interactive proof system TLAPS, completeness is not an issue: Whenever type inference fails, the user has to prove the obligation by hand. [MV14] also introduced refinement types for TLA$^+$, which fit well the task of mechanized theorem proving. We rather focus on decidable type inference, but for a less expressive type system.

[KKT19b] settle on a simple bottom-up type inference for operator-free expressions, which falls back to user annotations. Our work their type system in several ways: (1) we significantly expand the typable fragment of TLA$^+$ by including user-defined operators, and (2) we account for operator overloading and thus do not need user annotations.

Type inference for programming languages is, of course, a rich and mature field, so we cannot compare our work with the plethora of results in this field. We mention only the techniques that, in our understanding, are most related to our results. [DM82] present a syntax-based algorithm for computing principal type schemas, i.e., the most general types that can be assigned to functions, of which all monotypes types are instances. This approach unfortunately does directly apply to our setting. First, TLA$^+$ has limited operator overloading, which ruins the one-to-one correspondence between syntax and types. Second, TLA$^+$ expressions may update (global) state variables by constraining prime variable, e.g., $x' = 3$. This would probably require us to do an analysis of types that are similar to `ref`-types in OCaml.

The above seems to indicate that, rather than a bottom-up principal type inference system, it makes more sense to consider constraint-based approaches such as [Rém92, GS01]. However, we could not find a simple way of encoding operator overloading in these frameworks without introducing disjunctive constraints.

Our ideas of logic encoding are inspired by the work of [SS08], who introduce HM(X) type inference in the lenses of constraint-logic programming. However, instead of computing principal type schemas, we use them as a predefined knowledge about the built-in operators. We believe that this gives us a simpler set of constraints for type inference, which can be efficiently handled by SMT solvers. In our context, computing principal types would require an additional concretization step to extract a parameter-free type instance that could then be used.

Some of the problems that we experienced with TLA$^+$ also arise in the dynamically-typed languages. So one can think of applying gradual types [CS16]. Our setting is however

different in several aspects. First, although it might be productive to think of TLA$^+$ values as dynamically typed rather than untyped, it is not really the case. As we have shown in Section 8.2 some values can be used as different types at the same time (e.g., a function of a special shape is also a sequence). Second, TLA$^+$ specifications are not large (ranging from 100 to 2k lines of code). If the users are willing to annotate a specification with types, there is no strong motivation for the gradual introduction of types.

8.11 Discussions

Impact of imprecise record types. Owing to the mix of records in sets, our analysis approximates the set of fields that can be contained in a record. As we mentioned before, this analysis can be made more precise by post-processing, e.g., to split the fields of two record types that do not interact in any way. One of the goals of our type inference is to distinguish records from other function-like values. Once this is done, it is much easier to write better type analysis. We see the main applications of our result in either translating TLA$^+$ into SMT constraints for symbolic model checking, or translating TLA$^+$ in an executable specification in a programming language. In both scenarios, it is easy to add a dynamic test of whether a field belongs to a record. (However, without our result, it is quite hard to implement all combinations of operators applied to all combinations of values.)

Initially, we considered sum types as an alternative to our current approach of approximating the set of record fields. As there are too many ways of testing for the actual record shape in TLA$^+$, we did not consider the approach with the sum types feasible. For the future work, we are considering to introduce a TLA$^+$ module that would allow the users to use records as algebraic data types. If the users would use this module, we would make our analysis more precise.

Type errors. If a specification e is ill-typed, the produced logic constraint $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$ will be unsatisfiable. For instance, there is no type assignment in the expression $1 + "a"$. In this case, the user would expect an error message that explains the possible source of type errors. Our approach to this is as follows. Whenever the tool generates a constraint of the form $[\![e]\!]^{\mathsf{TLA}}_{\blacktriangleleft t}$, it associates t with the position of e in the specification source file (line number and offset). If the SMT solver determines that the produced constraints are unsatisfiable, it gives us an unsatisfiability core, that is, a small number of assertions that alone lead to contradiction. By computing the SMT variables appearing in the unsatisfiability core and mapping them to the source locations, we give the user the regions in the source file that produce type-conflicting constraints.

Principal types. Our approach only computes a collection of monotypes for user-defined operators. We believe that there are two orthogonal ways to partially lifting our results to computing principal types. The first approach is using MaxSMT: Instead of pure SMT, we can use a MaxSMT optimization mode as in, e.g., [HUEM18]. MaxSMT

is implemented in Z3 as well [BPF15]. In this case, we add one soft constraint per variable that is introduced for each expression in a specification. These soft constraints maximize the reward for the solver when a type is treated as a type variable. We have implemented this approach and did preliminary experiments. In the few experiments we manually inspected, the computed types were the expected principal types. However, we felt that these results are too preliminary to report about. The second approach is in using post-generalization. Assume that a user-defined operator F has two computed monotypes $\langle \tau_1, \ldots, \tau_n \rangle \Rightarrow \hat{\tau}$ and $\langle \rho_1, \ldots, \rho_n \rangle \Rightarrow \hat{\rho}$. If, for example, it is neither the case that τ_1 is a subtype of ρ_1, nor ρ_1 is a subtype of τ_1, then we can apply unification to introduce a type variable in the first argument. Again, we inspected the generalized type for our benchmarks and the results look promising, but we have not investigated the theoretical limits of this approach yet.

8.12 Conclusions

Our experiments demonstrate the feasibility of our type-inference approach over a large set of benchmarks. Though there are some bottlenecks in the largest of specifications, many improvements could still be made to the implementation to bring down runtime to a reasonable level in those cases. Overall, we believe our type inference, in addition to existing tools, such as model checkers, provides users with a variety of options for getting automatic feedback about their specifications.

We had to address many idiosyncrasies of TLA$^+$. Interestingly, some of them allowed us to come with a conceptually simple approach. By treating almost all of the language constructs as operators, we minimize the number of translation rules. Somewhat unexpectedly, the following language restrictions allow us to transform specifications into a conceptually simple format: absence of currying, inability to return operators as value, inability to pass higher-order operators as arguments to other higher-order operators. For instance, it makes it possible to clone operator definitions at application sites (see Section 8.8.1).

8.13 Schemas of all built-in operators

For completness, we list the schemas of all built-in operators in this section.

8.13.1 Operators with primitive schemas

Tables 8.9 to 8.18 list the TLA$^+$ operators with primitive schemas, grouped by category.

8.13.2 Overloaded operators

Compared to operators with primitive schemas, operators with complex schemas are much fewer in number.

Access with [_]: We distinguish four different primitive schemas for the application operator; one for functions, one of records, one for tuples, and one for sequences. Based on static analysis of the argument, up to three of them are simultaneously possible. Denote by s_1 the schema $\forall\alpha_1, \alpha_2 \,.\, \langle \alpha_1 \to \alpha_2, \alpha_1 \rangle \Rightarrow \alpha_2$, by s_2 the schema $\forall\alpha_3 \,.\, \langle \mathsf{Seq}(\alpha_3), \mathsf{Int} \rangle \Rightarrow \alpha_3$, by $s_3(k)$ the schema $\forall\alpha_4 \,.\, \langle \langle k \mapsto \alpha_4 \rangle, \mathsf{Int} \rangle \Rightarrow \alpha_4$ and by $s_4(h)$ the schema $\forall\alpha_5 \,.\, \langle [h \mapsto \alpha_5], \mathsf{String} \rangle \Rightarrow \alpha_5$. We distinguish the following cases, for an application $e_1[e_2]$:

- If e_2 is an integer literal k, then [_] has the schema $s_1 \sqcup s_2 \sqcup s_3(k)$
- If e_2 is a string literal h, then [_] has the schema $s_1 \sqcup s_4(h)$
- Otherwise, [_] has the schema $s_1 \sqcup s_2$

This is because tuples (resp. records) may only be accessed at statically known indices (resp. fields), since different fields of a tuple (resp. record) may hold different types.

Modification with except: The EXCEPT operator may be used to modify functions, sequences or records. Denote by s_1 the schema $\forall\alpha_1, \alpha_2 \,.\, \langle \alpha_1 \to \alpha_2, \alpha_1, \alpha_2 \rangle \Rightarrow \alpha_1 \to \alpha_2$, by s_2 the schema $\forall\alpha_3 \,.\, \langle \mathsf{Seq}(\alpha_3), \mathsf{Int}, \alpha_3 \rangle \Rightarrow \mathsf{Seq}(\alpha_3)$ and by $s_3(h)$ the schema $\forall\alpha_4 \,.\, \langle [h \mapsto \alpha_4], \mathsf{String}, \alpha_4 \rangle \Rightarrow [h \mapsto \alpha_4]$. We distinguish the following cases, for an application $[e_1 \text{ EXCEPT } ![e_2] = e_3]$:

- If e_2 is a string literal h, then EXCEPT has the schema $s_1 \sqcup s_3(h)$
- Otherwise, EXCEPT has the schema $s_1 \sqcup s_2$

Constructor $\langle \_, \ldots, \_ \rangle_n$: This constructor may be used to define both tuples and sequences. Denote by s_1 the schema $\forall\alpha_1, \ldots, \alpha_n \,.\, \langle \alpha_1, \ldots, \alpha_n \rangle \Rightarrow \langle \alpha_1, \ldots, \alpha_n \rangle$ and by s_2 the schema $\forall\hat{\alpha} \,.\, \underbrace{\langle \hat{\alpha}, \ldots, \hat{\alpha} \rangle}_{n-\text{times}} \Rightarrow \mathsf{Seq}(\hat{\alpha})$. Then, $\langle \_, \ldots, \_ \rangle_n$ has the schema $s_1 \sqcup s_2$.

Domain operator domain: We distinguish four different primitive schemas for the DOMAIN operator; one for functions, one of records, one for tuples, and one for sequences. We refer to the empty record type [] as τ_r and to the empty sparse-tuple type $\langle\rangle$ as τ_{st}. Denote by s_1 the schema $\forall\alpha_1, \alpha_2 \,.\, \langle \alpha_1 \to \alpha_2 \rangle \Rightarrow \mathsf{Set}(\alpha_1)$, by s_2 the schema $\langle \tau_r \rangle \Rightarrow \mathsf{Set}(\mathsf{String})$, by s_3 the schema $\langle \tau_{st} \rangle \Rightarrow \mathsf{Set}(\mathsf{Int})$ and by s_4 the schema $\forall\alpha_3 \,.\, \langle \mathsf{Seq}(\alpha_3) \rangle \Rightarrow \mathsf{Set}(\mathsf{Int})$. Then, DOMAIN has the schema $s_1 \sqcup s_2 \sqcup s_3 \sqcup s_4$. The role of the empty record type and empty sparse-tuple type is to act as a catch-all, since τ_r is a subtype of all record types and τ_{st} is a subtype of all sparse-tuple types an all fixed-size types (for all sizes).

Table 8.8: A list of all specification IDs used in our experiments

ID	Specification
1	Stones
2	StonesFix
3	TransactionCommit
4	DieHard
5	CarTalkPuzzle
6	MissionariesAndCannibals
7	byihive-VoucherLifeCycle
8	Queens
9	QueensFix
10	SpanningTree
11	MisraReachability
12	SimpleAllocator
13	Spanning
14	nbaccRay97Fix
15	nbaccRay97
16	Paxos
17	EWD840
18	ChangRoberts
19	bcastByz
20	FPaxos
21	SumsEven
22	bcastFolklore
23	LamportMutex
24	AbaAsynByz
25	AbaAsynByzFix
26	2PCwithBTM
27	cf1sFolklore
28	DijkstraMutex
29	DijkstraMutexFix
30	Bosco
31	CbcMax
32	nbacgGuer01
33	byihive-VoucherIssue
34	byihive-VoucherTransfer
35	byihive-VoucherCancel
36	byihive-VoucherRedeem
37	c1cs
38	Bakery
49	RaftFix
40	Raft
41	802.16-AuthorizationPKMv35
42	802.16-AuthorizationPKMv35Fix

Table 8.9: Logic operators

Operator(s)	Schema
$=, \neq$	$\forall \alpha \ . \ \langle \alpha, \alpha \rangle \Rightarrow \mathsf{Bool}$
$\wedge, \vee, \Rightarrow, \equiv$	$\langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$\neg$	$\langle \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$\exists_{\text{bounded}}, \forall_{\text{bounded}}$	$\forall \alpha \ . \ \langle \alpha, \mathsf{Set}(\alpha), \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$\exists_{\text{unbounded}}, \forall_{\text{unbounded}}$	$\forall \alpha \ . \ \langle \alpha, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$\text{CHOOSE}_{\text{bounded}}$	$\forall \alpha \ . \ \langle \alpha, \mathsf{Set}(\alpha), \mathsf{Bool} \rangle \Rightarrow \alpha$
$\text{CHOOSE}_{\text{unbounded}}$	$\forall \alpha \ . \ \langle \alpha, \mathsf{Bool} \rangle \Rightarrow \alpha$

Table 8.10: Arithmetic operators

Operator(s)	Schema
$+, -, \times, /, \%$	$\langle \mathsf{Int}, \mathsf{Int} \rangle \Rightarrow \mathsf{Int}$
$-_{\text{unary}}$	$\langle \mathsf{Int} \rangle \Rightarrow \mathsf{Int}$
$..$	$\langle \mathsf{Int}, \mathsf{Int} \rangle \Rightarrow \mathsf{Set}(\mathsf{Int})$
$<, >, \leq, \geq$	$\langle \mathsf{Int}, \mathsf{Int} \rangle \Rightarrow \mathsf{Bool}$

Table 8.11: Set operators

Operator(s)	Schema
$\in \ , \ \notin$	$\forall \alpha \ . \langle \alpha, \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Bool}$
$\subset, \supset, \subseteq, \supseteq$	$\forall \alpha \ . \langle \mathsf{Set}(\alpha), \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Bool}$
$\cup, \cap, \setminus$	$\forall \alpha \ . \langle \mathsf{Set}(\alpha), \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Set}(\alpha)$
$\{\_\}_n, n \in \mathbb{N}_0$	$\forall \alpha \ . \underbrace{\langle \alpha, \ldots, \alpha \rangle}_{n-\text{times}} \Rightarrow \mathsf{Set}(\alpha)$
$\{\_ \in \_ : \_\}$	$\forall \alpha \ . \langle \alpha, \mathsf{Set}(\alpha), \mathsf{Bool} \rangle \Rightarrow \mathsf{Set}(\alpha)$
$\{\_ : \_ \in \_\}$	$\forall \alpha_1, \alpha_2 \ . \langle \alpha_2, \alpha_1, \mathsf{Set}(\alpha_1) \rangle \Rightarrow \mathsf{Set}(\alpha_2)$
SUBSET	$\forall \alpha \ . \langle \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Set}(\mathsf{Set}(\alpha))$
UNION	$\forall \alpha \ . \langle \mathsf{Set}(\mathsf{Set}(\alpha)) \rangle \Rightarrow \mathsf{Set}(\alpha)$
Cardinality	$\forall \alpha \ . \ \langle \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Int}$
IsFiniteSet	$\forall \alpha \ . \ \langle \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Bool}$

Table 8.12: Action operators

Operator(s)	Schema
ENABLED	$\langle \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$[\_]\_, \langle \_ \rangle \_$	$\forall \alpha \ . \langle \mathsf{Bool}, \alpha \rangle \Rightarrow \mathsf{Bool}$
$\_ \cdot \_$	$\langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$

Table 8.13: Temporal operators

Operator(s)	Schema
$\Box, \Diamond$	$\langle \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$\text{WF}\_(\_), \text{SF}\_(\_), \mathbf{\exists}, \mathbf{\forall}$	$\forall \alpha \ . \ \langle \alpha, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$
$\rightsquigarrow, \overset{+}{\rightarrow}$	$\langle \mathsf{Bool}, \mathsf{Bool} \rangle \Rightarrow \mathsf{Bool}$

Table 8.14: Function operators

Operator(s)	Schema
$[\_ \to \_]$	$\forall \alpha_1, \alpha_2 \,.\, \langle \mathsf{Set}(\alpha_1), \mathsf{Set}(\alpha_2) \rangle \Rightarrow \mathsf{Set}(\alpha_1 \to \alpha_2)$
$[\_ \in \_ \mapsto \_]$	$\forall \alpha_1, \alpha_2 \,.\, \langle \alpha, \mathsf{Set}(\alpha), \alpha_2 \rangle \Rightarrow \alpha_1 \to \alpha_2$

Table 8.15: Record operators, $n \in \mathbb{N}$

Operator(s)	Schema
$[h_1 \mapsto \_, \ldots, h_n \mapsto \_]$	$\forall \alpha_1, \ldots, \alpha_n \,.\, \langle \alpha_1, \ldots, \alpha_n \rangle \Rightarrow [h_1 \mapsto \alpha_1, \ldots, h_n \mapsto \alpha_n]$
$[h_1 : \_, \ldots, h_n : \_]$	$\forall \alpha_1, \ldots, \alpha_n \,.\, \langle \mathsf{Set}(\alpha_1), \ldots, \mathsf{Set}(\alpha_n) \rangle \Rightarrow \mathsf{Set}([h_1 : \alpha_1, \ldots, h_n \mapsto \alpha_n])$

Table 8.16: Tuple operators, $n \geq 2$

Operator(s)	Schema
$\times_n$	$\forall \alpha_1, \ldots, \alpha_n \,.\, \langle \mathsf{Set}(\alpha_1), \ldots, \mathsf{Set}(\alpha_n) \rangle \Rightarrow \mathsf{Set}(\langle \alpha_1, \ldots, \alpha_n \rangle)$

Table 8.17: Sequence operators

Operator(s)	Schema
$\circ$	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha), \mathsf{Seq}(\alpha) \rangle \Rightarrow \mathsf{Seq}(\alpha)$
Head	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha) \rangle \Rightarrow \alpha$
Tail	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha) \rangle \Rightarrow \mathsf{Seq}(\alpha)$
Len	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha) \rangle \Rightarrow \mathsf{Int}$
Seq	$\forall \alpha \,.\, \langle \mathsf{Set}(\alpha) \rangle \Rightarrow \mathsf{Set}(\mathsf{Seq}(\alpha))$
Append	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha), \alpha \rangle \Rightarrow \mathsf{Seq}(\alpha)$
SubSeq	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha), \mathsf{Int}, \mathsf{Int} \rangle \Rightarrow \mathsf{Seq}(\alpha)$
SelectSeq	$\forall \alpha \,.\, \langle \mathsf{Seq}(\alpha), \langle \alpha \rangle \Rightarrow \mathsf{Bool} \rangle \Rightarrow \mathsf{Seq}(\alpha)$

Table 8.18: Control operators, $n \in \mathbb{N}$

Operator(s)	Schema
IF $\_$ THEN $\_$ ELSE $\_$	$\forall \alpha \,.\, \langle \mathsf{Bool}, \alpha, \alpha \rangle \Rightarrow \alpha$
CASE $\underbrace{\_ \to \_ \,\Box \ldots \Box\, \_ \to \_}_{n-\text{times}}$	$\forall \alpha \,.\, \underbrace{\langle \mathsf{Bool}, \alpha, \ldots, \mathsf{Bool}, \alpha \rangle}_{n-\text{times}} \Rightarrow \alpha$
CASE $\underbrace{\_ \to \_ \,\Box \ldots \Box\, \_ \to \_}_{n-\text{times}}$ $\Box$ OTHER $\to \_$	$\forall \alpha \,.\, \left\langle \underbrace{\mathsf{Bool}, \alpha, \ldots, \mathsf{Bool}, \alpha}_{n-\text{times}}, \alpha \right\rangle \Rightarrow \alpha$

CHAPTER 9

Conclusions

In this thesis, we have presented the various steps required to translate a TLA$^+$ specification into SMT constraints, solve them using an SMT solver, and use the resulting proof or counterexample to to reason about the original specification and its (inductive) invariants. We've demonstrated that these methods may be implemented in a robust tool, Apalache, which can be used to analyze specifications of state-of-the-art distributed systems and protocols. These benchmarks make use of the full expressiveness of the TLA$^+$ language, including features like nested sets, records, functions, and more. Our symbolic approach using SMT is novel, and offers many benefits compared to the previous standard tool TLC, which utilized explicit-state model checking, which is known to have issues in scalability with an inherent state-space explosion for select language feature combinations.

We have defined rewriting rules for individual TLA$^+$ operators in Chapter 7, which allow us to represent applications of those operators in SMT either in a vacuum or as part of a larger specification. We have utilized auxiliary structures called *arenas*, which facilitate this translation, by keeping track of data structure overapproximations. In combination with an SMT model, the arena overapproximations concretize to describe exact TLA$^+$ values in e.g. a counterexample trace.

Additionally, Chapter 8 outlines a type system for TLA$^+$, as well as a means of type inference. While types are not originally present in the language itself, they are a common feature in many programming- and specification languages, as well as a pragmatically useful tool for specification authors. While Apalache requires types for its encoding construction, the type system and typechecking are independently useful, according to testimonies of specification authors, who use tools besides Apalache (e.g. TLC).

9.1 Future Work

Alternate SMT Theories

The SMT translation presented in this thesis is general-purpose, and aims to capture as broad of a fragment of TLA$^+$ as possible. As such, it is very likely, that there exist, possibly conditional, translations of certain language features which exhibit better performance. For instance, the SMT theory of arrays appears to be better suited for some constructs, as we have investigated in [OKK+23], but there are numerous other theories and constructs that could use specialized translations.

Restricted fragments of TLA$^+$

If we are allowed to reject certain commonly-used features of the language, we can also seek to create a more straightforward SMT encoding. Consider, for instance, sets in TLA$^+$. In many specifications, sets are only used as generic containers; one merely needs to be able to add and remove elements, and query membership. For those specifications, we can observe a semantic duality between sets and functions: for every set $S : Set(T)$, we can define a function $F_S : \mathbb{T} \to \text{BOOLEAN}$, where $\mathbb{T}$ is the set of all values of type T (e.g. Int), such that $F_S(t) = \text{TRUE} \iff t \in S$. We get nice properties, such as $F_{S \cup R}(t) = F_S(t) \vee F_R(t)$, $F_{S \cap R}(t) = F_S(t) \wedge F_R(t)$, $F_{\{x \in S : P\}}(t) = F_S(t) \wedge P(t)$. Due to this, it is possible to rewrite many (though not all) specifications, such that they are semantically equivalent, but do not use sets (or e.g. tuples/sequences).

Since sets are the main reason why our translation introduces arenas, we can actually define a fragment of TLA$^+$, which would translate directly to the `QF_UF` fragment (possibly with the addition of arithmetic), without the need for arenas. Alternatively, such a fragment could also be transpiled to Ivy, giving us access to the full breadth of Ivy tooling. This would, for example, allow us to use Ivy's executable code generator, to compile TLA$^+$ specifications to code.

It remains to be seen whether an automatic syntactic translation from a fragment of TLA$^+$, in with sets can be equivalently replaced with functions as described above, to this set-free fragment is practical, but this step could potentially be automated as well.

Bibliography

[ACK+02] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[all21] Alloy case studies. `https://alloytools.org/citations/case-studies.html`, 2021.

[ALN+91] J. R. Abrial, M. K. O. Lee, D. S. Neilson, P. N. Scharbach, and I. H. Sørensen. The b-method. In Søren Prehn and Hans Toetenel, editors, *VDM '91 Formal Software Development Methods*, pages 398–405, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[AMW16] Noran Azmy, Stephan Merz, and Christoph Weidenbach. A rigorous correctness proof for pastry. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 86–101. Springer, 2016.

[AMW18] Noran Azmy, Stephan Merz, and Christoph Weidenbach. A machine-checked correctness proof for pastry. *Sci. Comput. Program.*, 158:64–80, 2018.

[Avi76] A Aviziens. Fault-tolerant systems. *IEEE Transactions on Computers*, 100(12):1304–1312, 1976.

[AZ02] Jeremy Avigad and Richard Zach. The epsilon calculus. 2002.

[Bar77] Jon Barwise. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*, volume 90, pages 5–46. Elsevier, 1977.

[BAT14] Gavin Bierman, Martín Abadi, and Mads Torgersen. Understanding typescript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.

[BBC+19] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. Reachability analysis for AWS-based networks. In *International Conference on Computer Aided Verification*, pages 231–241. Springer, 2019.

[BBP13] Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning*, 51(1):109–128, 2013.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *TACAS*, volume 1579 of *LNCS*, pages 193–207, 1999.

[BCD+05] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *LPAR*, volume 4790, pages 151–165. Springer, 2007.

[ben] Benchmark repository. `https://github.com/konnov/fault-tolerant-benchmarks/tree/master/2015`.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[BGMR01] Francisco Vilar Brasileiro, Fabíola Greve, Achour Mostéfaoui, and Michel Raynal. Consensus in one communication step. In *PaCT*, volume 2127 of *LNCS*, pages 42–50, 2001.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[BJK+15] Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. *Decidability of Parameterized Verification*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2015.

[BKM18] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on bft consensus. *arXiv preprint arXiv:1807.04938*, 2018.

[BLL+19] Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. Verification of threshold-based distributed algorithms by decomposition to decidable logics. In *CAV*, pages 245–266, 2019.

[BLS04] Mike Barnett, K Rustan M Leino, and Wolfram Schulte. The spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 49–69. Springer, 2004.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *PLDI*, pages 203–213, 2001.

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - an optimizing SMT solver. In *TACAS*, pages 194–199, 2015.

[BSM16] Srijita Basu, Anirban Sengupta, and Chandan Mazumdar. Modelling operations and security of cloud systems using z-notation and chinese wall security policy. *Enterprise Information Systems*, 10(9):1024–1046, 2016.

[Buc16] Ethan Buchman. Tendermint: Byzantine fault tolerance in the age of Blockchains. Master's thesis, University of Guelph, 2016. `http://hdl.handle.net/10214/9769`.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *International Conference on Computer Aided Verification*, pages 334–342. Springer, 2014.

[CCG+02] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[CCLQ97] Gianpiero Cabodi, Paolo Camurati, Luciano Lavagno, and Stefano Quer. Disjunctive partitioning and partial iterative squaring: an effective approach for symbolic traversal of large circuits. In *Proceedings of the 34th annual Design Automation Conference*, pages 728–733. ACM, 1997.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla+ proofs. In *International Symposium on Formal Methods*, pages 147–154. Springer, 2012.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. The TLA$^+$ proof system: Building a heterogeneous verification platform. In *Theoretical aspects of computing*, pages 44–44. Springer-Verlag, 2010.

[CDP+17] Mathieu Comptier, David Deharbe, Julien Molinero Perez, Louis Mussat, Thibaut Pierre, and Denis Sabatier. Safety analysis of a cbtc system: A rigorous approach with event-b. In Alessandro Fantechi, Thierry Lecomte, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 148–159, Cham, 2017. Springer International Publishing.

[CEJS98] Edmund M Clarke, E Allen Emerson, Somesh Jha, and A Prasad Sistla. Symmetry reductions in model checking. In *International Conference on Computer Aided Verification*, pages 147–158. Springer, 1998.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[CHVB18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*, volume 10. Springer, 2018.

[Cie97] Krzysztof Ciesielski. *Set theory for the working mathematician*. Number 39. Cambridge University Press, 1997.

[CL98] Ernie Cohen and Leslie Lamport. Reduction in TLA. In *CONCUR*, LNCS, pages 317–331, 1998.

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In *International Conference on Computer Aided Verification*, pages 510–517. Springer, 2016.

[CP09] Alcino Cunha and Hugo Pacheco. Mapping between alloy specifications and database implementations. In *2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, pages 285–294. IEEE, 2009.

[CR16] Maximiliano Cristiá and Gianfranco Rossi. A decision procedure for sets, binary relations and partial functions. In *CAV*, pages 179–198, 2016.

[CS16] Matteo Cimini and Jeremy G. Siek. The gradualizer: a methodology and algorithm for generating gradual type systems. In *POPL*, pages 443–455, 2016.

[DB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 1579 of *LNCS*, pages 337–340. 2008.

[DDMW19] Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. Communication-closed asynchronous protocols. In *CAV*, pages 344–363, 2019.

[DHV+14] Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. A logic-based framework for verifying consensus algorithms. In *VMCAI*, volume 8318 of *LNCS*, pages 161–181, 2014.

[DHZ16] Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*, pages 400–415, 2016.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, 1982.

[DRK+14] Morgan Deters, Andrew Reynolds, Tim King, Clark Barrett, and Cesare Tinelli. A tour of cvc4: How it works, and how to use it. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 7–7, 2014.

[DS06] Dan Dobre and Neeraj Suri. One-step consensus with zero-degradation. In *DSN*, pages 137–146, 2006.

[DTT14] Giorgio Delzanno, Michele Tatarek, and Riccardo Traverso. Model checking paxos in spin. In *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014.*, pages 131–146, 2014.

[EGL92] Urban Engberg, Peter Grønning, and Leslie Lamport. Mechanical verification of concurrent systems with tla. In *International Conference on Computer Aided Verification*, pages 44–55. Springer, 1992.

[EMT+17] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *International Conference on Computer Aided Verification*, pages 126–133. Springer, 2017.

[erc] A TLA+ specification of erc20. `https://github.com/informalsystems/tla-apalache-workshop/blob/main/examples/erc20-approve-attack/ERC20.tla`. [Online; accessed 03-March-2022].

[FKP16] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Proving liveness of parameterized programs. In *LICS*, pages 185–196, 2016.

[Gar81] Lynn E Garner. On the collatz 3n+ 1 algorithm. *Proceedings of the American Mathematical Society*, 82(1):19–22, 1981.

[GL03a] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[GL03b] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[GL06] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, 2006.

[GS01] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *Programs as Data Objects (PADO)*, pages 63–83, 2001.

[Gue02] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15(1):17–25, 2002.

[Gus19] Jason Gustafson. Kafka improvement proposal 320, 2019.

[HB07] Gerard J Holzmann and Dragan Bosnacki. Multi-core model checking with spin. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[HBL14] Dominik Hansen, Jens Bendisposto, and Michael Leuschel. Integrating prob into the tla toolbox. 2014.

[HHK+17] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, June 2017.

[HL96] Howard Haughton and Kevin Lano. *Specification in B: An introduction using the B toolkit*. World Scientific, 1996.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hol03] Gerard Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[HUEM18] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. MaxSMT-based type inference for Python 3. In *CAV*, pages 12–19, 2018.

[ivy] Ivy github repository. `http://microsoft.github.io/ivy/`.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):256–290, 2002.

[JLM+03] Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking cache-coherence protocols with tla+. *Formal Methods in System Design*, 22(2):125–131, 2003.

[JSS00] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the alloy constraint analyzer. In *Proceedings of the 22nd international conference on Software engineering*, pages 730–733. ACM, 2000.

[Kel76] Robert M Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[KGS12] Arvinder Kaur, Samridhi Gulati, and Sarita Singh. A comparative study of two formal specification languages: Z-notation & b-method. In *Proceedings of the Second International Conference on Computational Science, Engineering and Information Technology*, CCSEIT '12, page 524–531, New York, NY, USA, 2012. Association for Computing Machinery.

[KK73] Ervin Knuth Knuth and Donald Ervin Knuth. *The art of computer programming: sorting and searching*, volume 3. Addison-Wesley/Helix Books, 1973.

[KK20] Israel Koren and C Mani Krishna. *Fault-tolerant systems*. Morgan Kaufmann, 2020.

[KKKF20] Igor Konnov, Jure Kukovec, Andrey Kuprianov, and Shon Feder. Apalache manual. `https://apalache.informal.systems/docs/apalache/index.html`, 2020.

[KKM22] Igor Konnov, Markus Kuppe, and Stephan Merz. Specification and verification with the tla+ trifecta: Tlc, apalache, and tlaps. In *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles: 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22–30, 2022, Proceedings, Part I*, pages 88–105. Springer, 2022.

[KKT19a] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. Apalache model checker. `https://github.com/konnov/apalache`, 2019.

[KKT19b] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA):123:1–123:30, 2019.

[KKW18] Jure Kukovec, Igor Konnov, and Josef Widder. Reachability in parameterized systems: all flavors of threshold automata. In *CONCUR 2018-29th International Conference on Concurrency Theory*, 2018.

[KLR19a] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The tla+ toolbox. *Electronic Proceedings in Theoretical Computer Science*, 310:50–62, Dec 2019.

[KLR19b] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ toolbox. In *Proceedings Fifth Workshop on Formal Integrated Development*

Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019, pages 50–62, 2019.

[KLSW20] Igor Konnov, Marijana Lazić, Ilina Stoilkovska, and Josef Widder. Tutorial: Parameterized verification with byzantine model checker. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 189–207. Springer, 2020.

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

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

[KNR05] Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. An algorithm for deciding BAPA: boolean algebra with presburger arithmetic. In *CADE*, pages 260–277, 2005.

[KTK18] Jure Kukovec, Thanh-Hai Tran, and Igor Konnov. Extracting symbolic transitions from TLA+ specifications. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 89–104. Springer, 2018.

[KTK20] Jure Kukovec, Thanh-Hai Tran, and Igor Konnov. Extracting symbolic transitions from tla$^+$ specifications. *Sci. Comput. Program.*, 187:102361, 2020.

[Kup17] Markus A Kuppe. *A Verified and Scalable Hash Table for the TLC Model Checker: Towards an Order of Magnitude Speedup*. PhD thesis, Master's thesis. University of Hamburg. http://www. lemmster. de/talks . . . , 2017.

[KVW14] Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. In *CONCUR*, volume 8704, pages 125–140. 2014.

[KVW15] Igor Konnov, Helmut Veith, and Josef Widder. SMT and POR beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *CAV (Part I)*, volume 9206 of *LNCS*, pages 85–102, 2015.

[KVW17] Igor Konnov, Helmut Veith, and Josef Widder. On the completeness of bounded model checking for threshold-based distributed algorithms: Reachability. *Information and Computation*, 252:95–109, 2017.

[KW18] Igor Konnov and Josef Widder. ByMC: Byzantine model checker. In *ISoLA (3)*, volume 11246 of *LNCS*, pages 327–342. Springer, 2018.

[L+01] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lam99] Leslie Lamport. Specifying concurrent systems with tlaˆ+. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:183–250, 1999.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Lam18] Leslie Lamport. TLA+2: A preliminary guide, 2018.

[LB03] Michael Leuschel and Michael Butler. Prob: A model checker for b. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, pages 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[Lec14] Thierry Lecomte. Atelier b. *Formal Methods Applied to Complex Systems: Implementation of the B Method*, pages 35–46, 2014.

[Lei08] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131):9, 2008.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.

[Lip75] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[LS79] Butler Lampson and Howard E Sturgis. Crash recovery in a distributed data storage system. 1979.

[Lyn96] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[MAK13] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *SOSP*, pages 358–372. ACM, 2013.

[MC16] Nuno Macedo and Alcino Cunha. Alloy meets TLA+: An exploratory study. *arXiv preprint arXiv:1603.03599*, 2016.

[McM93] Kenneth L McMillan. The SMV system. In *Symbolic Model Checking*, pages 61–85. Springer, 1993.

[Mer08a] Stephan Merz. The specification language TLA$^+$. In *Logics of specification languages*, pages 401–451. Springer, 2008.

[Mer08b] Stephan Merz. The specification language TLA$^+$. In Dines Bjørner and Martin C. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 401–451. Springer, Berlin-Heidelberg, 2008.

[Mer12] Stephan Merz. On the logic of TLA$^+$. *Computing and Informatics*, 22(3-4):351–379, 2012.

[Min67] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[MMPR03] Achour Mostéfaoui, Eric Mourgaya, Philippe Raipin Parvédy, and Michel Raynal. Evaluating the condition-based approach to solve consensus. In *DSN*, pages 541–550, 2003.

[MP20] Kenneth L McMillan and Oded Padon. Ivy: a multi-modal verification tool for distributed algorithms. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, pages 190–202. Springer, 2020.

[MRTB17] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Relational constraint solving in SMT. In *International Conference on Automated Deduction*, pages 148–165. Springer, 2017.

[MSB17] Ognjen Maric, Christoph Sprenger, and David A. Basin. Cutoff bounds for consensus algorithms. In *CAV, Part II*, pages 217–237, 2017.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*, pages 696–701. Springer, 2013.

[MV12a] Stephan Merz and Hernán Vanzetto. Automatic verification of TLA$^+$ proof obligations with SMT solvers. In *LPAR*, volume 7180, pages 289–303. Springer, 2012.

[MV12b] Stephan Merz and Hernán Vanzetto. Harnessing SMT solvers for TLA$^+$ proofs. *ECEASST*, 53, 2012.

[MV14] Stephan Merz and Hernán Vanzetto. Refinement types for TLA$^+$. In *NASA Formal Methods Symposium*, pages 143–157. Springer, 2014.

[MV18] Stephan Merz and Hernán Vanzetto. Encoding TLA+ into unsorted and many-sorted first-order logic. *Science of Computer Programming*, 158:3–20, 2018.

[Nak] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. URL: `https://bitcoin.org/bitcoin.pdf`.

[New14] Chris Newcombe. Why Amazon chose TLA$^+$. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[NGHS17] Michael Nofer, Peter Gomber, Oliver Hinz, and Dirk Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[NRZ+15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon web services uses formal methods. *Comm. ACM*, 58(4):66–73, 2015.

[OKK+23] Rodrigo Otoni, Igor Konnov, Jure Kukovec, Patrick Eugster, and Natasha Sharygina. Symbolic model checking for tla+ made faster. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 126–144, Cham, 2023. Springer Nature Switzerland.

[Ong14] Diego Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford U., 2014.

[OPSR15] Francisco Ortin, Jose Baltasar Garcia Perez-Schofield, and Jose Manuel Redondo. Towards a static type checker for python. In *European Conference on Object-Oriented Programming (ECOOP), Scripts to Programs Workshop, STOP*, volume 15, pages 1–2, 2015.

[P+10] Vijay Pande et al. Folding@ home. *Distributed Computing*, 2010.

[Pie04] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[PLSS17] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL*, 1(OOPSLA):108:1–108:31, 2017.

[PMP+16] Oded Padon, Kenneth L McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 614–630, 2016.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[pro] WWTF project ICT15-103 APALACHE. `https://forsyte.at/research/apalache/`. [Online; accessed 4-Feb-2018].

[PS07] Lawrence C Paulson and Kong Woei Susanto. Source-level proof reconstruction for interactive theorem proving. In *International Conference on Theorem Proving in Higher Order Logics*, pages 232–245. Springer, 2007.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[Ray97] Michel Raynal. A case study of agreement problems in distributed systems: Non-blocking atomic commitment. In *HASE*, pages 209–214, 1997.

[Rém92] Didier Rémy. Extending ml type system with a sorted equational theory. Technical report, Research Report 1766, Institut National de Recherche en Informatique et Automatisme, 1992.

[RGBC17] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Sci. Comput. Program.*, 148:26–48, 2017.

[Rob97] Ken Robinson. The B method and the B toolkit. In *International Conference on Algebraic Methodology and Software Technology*, pages 576–580. Springer, 1997.

[RSF+15] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for TypeScript. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 167–180, 2015.

[SA17] Wen Su and Jean-Raymond Abrial. Aircraft landing gear system: approaches with Event-B to the modeling of an industrial system. *International Journal on Software Tools for Technology Transfer*, 19(2):141–166, 2017.

[SAF+04] David A Stainforth, Myles R Allen, David Frame, Jamie Kettleborough, Carl Christensen, Tolu Aina, and Matthew Collins. Climateprediction. net: a global community for research in climate physics. In *Environmental online communication*, pages 101–112. Springer, 2004.

[set] Seti@home. URL: `setiathome.berkeley.edu`.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. Dependent types and multi-monadic effects in f. In *ACM SIGPLAN Notices*, volume 51, pages 256–270. ACM, 2016.

[SS08] Martin Sulzmann and Peter J. Stuckey. HM(X) type inference is CLP(X) solving. *J. Funct. Program.*, 18(2):251–283, 2008.

[ST87] T.K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Dist. Comp.*, 2:80–94, 1987.

[SvR08] Yee Jiun Song and Robbert van Renesse. Bosco: One-step byzantine asynchronous consensus. In Gadi Taubenfeld, editor, *Distributed Computing*, pages 438–450, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[SWT18] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *PACMPL*, 2(POPL):28:1–28:30, 2018.

[Sys20] Informal Systems. Apalache web page. `https://apalache.informal.systems`, 2020.

[ten] Tendermint specifications. `https://github.com/tendermint/tendermint/tree/master/spec`. [Online; accessed 03-March-2022].

[tlaa] A collection of TLA$^+$ specifications. `https://github.com/tlaplus/Examples/`. [Online; accessed 21-October-2017].

[TLAb] TLA+ proof system. `https://tla.msr-inria.inria.fr/tlaps/content/Home.html`.

[TLA20] A collection of TLA+ specifications of varying complexities, 2020. Last accessed on May 8, 2020.

[TRBB18] Cesare Tinelli, Andrew Reynolds, Clark Barrett, and Kshitij Bansal. Reasoning with finite sets and cardinality constraints in SMT. *Logical Methods in Computer Science*, 14, 2018.

[Val96] Antti Valmari. The state explosion problem. In *Advanced Course on Petri Nets*, pages 429–528. Springer, 1996.

[vGBR16] Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*, pages 599–613, 2016.

[vGKB$^+$19] Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL*, 3(POPL):59:1–59:30, 2019.

[VKSB14] Michael M Vitousek, Andrew M Kent, Jeremy G Siek, and Jim Baker. Design and evaluation of gradual typing for Python. In *Proceedings of the 10th ACM Symposium on Dynamic languages*, pages 45–56, 2014.

[Way18] Hillel Wayne. *Practical TLA+: Planning Driven Development*. Apress, 2018.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*, pages 357–368, 2015.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[YPK10] Kuat Yessenov, Ruzica Piskac, and Viktor Kuncak. Collections, cardinalities, and relations. In *VMCAI*, pages 380–395, 2010.

[Zaf09] Nazir Ahmad Zafar. Formal specification and validation of railway network components using z notation. *IET software*, 3(4):312–320, 2009.

[Zav15] Pamela Zave. A practical comparison of alloy and spin. *Formal Aspects of Computing*, 27(2):239–253, 2015.

[Zha16] Brandon Zhang. Pgo: Corresponding a high-level formal specification with its implementation. *SOSP SRC*, page 3, 2016.

[ZXHW10] Weimin Zheng, Pengzhi Xu, Xiaomeng Huang, and Nuo Wu. Design a cloud storage platform for pervasive computing environments. *Cluster Computing*, 13(2):141–151, 2010.