FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

User-guided Predicate Abstraction of TLA+ Specifications

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

European Master in Computational Logic

eingereicht von

Thanh Hai Tran
Matrikelnummer 1428553

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Helmut Veith
Mitwirkung: Igor Konnov, PhD

Wien, 10. Mai 2016

Thanh Hai Tran	Helmut Veith

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

User-guided Predicate Abstraction of TLA+ Specifications

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

European Master in Computational Logic

by

Thanh Hai Tran
Registration Number 1428553

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Helmut Veith
Assistance: Igor Konnov, PhD

Vienna, 10th May, 2016

Thanh Hai Tran	Helmut Veith

Erklärung zur Verfassung der Arbeit

Thanh Hai Tran
Zur Spinnerin 53/4/18, Vienna 1100, Austria

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

Wien, 10. Mai 2016

Thanh Hai Tran

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my advisors Helmut Veith and Igor Konnov for their continuous support on completing my internship and my master studies, for all their useful advice, insightful ideas that always made a difference and for the financial support. I would also like to thank Josef Widder for many interesting discussions and many valuable comments.

I would like to thank the Joint Commission of the European Master's Program in Computational Logic for the opportunity to be a part of this master program and for the received financial support during my studies. I am very grateful to all the people from every partner university that are involved in the organization of this program, especially for their support regarding all the formalities and making the transition from one country to others as smooth as possible.

Last but not least, I want to thank my family: my parents and my brother for all their encouragements and enormous support during my studies. To all my friends back home, in Bolzano, Dresden, Lisbon and Vienna - thank you all for making the last three years so memorable for me.

Kurzfassung

TLA$^+$ ist eine Sprache zur Spezifikation und Verifikation sequenzieller, nebenläufiger und verteilter Systeme. Sie basiert auf den folgenden beiden mathematischen Konzepten:

i. der temporalen Logik der Aktionen (TLA) zur Charakterisierung von dynamischem Systemverhalten, und

ii. einer Variante der Zermelo-Fraenkel-Mengenlehre mit Auswahlaxiom (ZFC) zur Beschreibung von Datenstrukturen.

Diese mathematischen Aspekte machen TLA$^+$ zu einer einfach verwendbaren und flexiblen Sprache für die Spezifikation verschiedenster Systeme. Außerdem ermöglicht es das solide mathematische Fundament der Sprache, die Korrektheit eines Systems formal zu beweisen. Doch obwohl formale Beweise die verlässlichste Methode zur Korrektheitsüberprüfung von Systemen darstellen, ist es oft schwer derartige Beweise zu finden.

Um die formale Verifikation von TLA$^+$ -Spezifikationen zu automatisieren wurde deshalb der Modelchecker TLC für TLA$^+$ eingeführt, der sich dadurch auszeichnet, dass er die erreichbaren Zustände explizit aufzählt. TLA$^+$ und TLC werden erfolgreich zur Spezifikation und Überprüfung industriell genutzer Systeme verwendet, jedoch wird die praktische Verwendbarkeit von TLC dadurch eingeschränkt, dass er keine Abstraktionstechniken zur Komplexitätsreduktion von Modellen unterstützt. Aus diesem Grund gilt TLC derzeit als ein Werkzeug, das hauptsächlich zur Erkennung einfacher Fehler in TLA$^+$ -Designs dient.

Bei der sogenannten Prädikatabstraktion handelt es sich um eine Technik, welche Software-Modelchecking für Systeme mit großem (auch unendlichem) Zustandsraum ermöglicht, indem sie den Zustandsraum mithilfe intelligenter Methoden verkleinert. Dabei wird, ausgehend von einer Menge von Prädikaten über den Systemvariablen, automatisch ein vereinfachtes abstraktes Modell des ursprünglichen Systems generiert, dessen Prädikatwerte und Transitionsrelationen anschließend, unter Verwendung von SMT-Solvern (satisfiability modulo theories solver), überprüft werden.

In dieser Arbeit verbinden wir TLA$^+$ mit benutzergeführter Prädikatabstraktion. Dabei ergeben sich die folgenden drei grundlegenden Herausforderungen:

- Bei TLA$^+$ handelt es sich um eine ungetypte Sprache, wohingegen SMT-Solver automatische Beweiser mit Entscheidungsprozeduren für verschiedenste Theorien der mehrsortigen Prädikatenlogik sind.
- TLA$^+$ enthält wichtige Sprachkonstrukte für welche es keine Entsprechung in SMT-Lib – der Sprache für SMT-Solver – gibt.
- Transitionssysteme werden in TLA$^+$ als Formeln mit komplexer Struktur spezifiziert. Beispielsweise kann eine typische TLA$^+$ -Spezifikation aus hunderten Zeilen von Formeln in Prädikatenlogik ohne Sorten bestehen, welche etwa Mengenkonstrukte, Quantifizierung über Verbunde oder Tupel, Konstrukte für Funktionen oder den sogenannten CASE-Operator verwenden.

Das Hauptziel dieser Arbeit ist die Entwicklung eines neuen Modelcheckers für TLA$^+$ -Spezifikationen, der basierend auf einer gegebenen Menge von Prädikaten ein abstraktes Modell erzeugen und verifizieren kann. Dabei sind die folgenden drei Hauptresultate dieser Arbeit herauszustreichen:

i. Wir präsentieren eine korrekte Übersetzung eines Fragments von TLA$^+$ in mehrsortige Prädikatenlogik. Während der Übersetzung werden diejenigen Konstrukte, für welche es keine Entsprechung in SMT-Lib gibt, durch Termersetzungsregeln eliminiert. Durch die Verwendung von Heuristiken wird dabei die Übersetzung vereinfacht. Obwohl unser Fragment Einschränkungen von TLA$^+$ -Ausdrücken einführt, ist es aus unserer Sicht nach wie vor ausreichend flexibel und ausdrucksstark um eine große Klasse verteilter Algorithmen zu spezifizieren. Unsere Übersetzung ist eine Erweiterung des Ansatzes von Merz und Vanzetto.

ii. Wir entwickeln einen Modelchecker, der Prädikatabstraktion und die kürzlich eingeführte IC3-Modelchecking-Technik unterstützt. Anstatt die Transitionsrelationen auszurollen, generiert dieser Modelchecker einen inkrementellen Beweis für eine gewünschte Sicherheitseigenschaft. Unsere vorläufigen Experimente zeigen, dass ein IC3-basierter Modelchecker effizienter als ein NuSMV2-basierter Modelchecker ist, allerdings hat ein IC3-Algorithmus den Nachteil, dass er keine Lebendigkeitseigenschaften testen kann.

Abstract

TLA$^+$ is a language for specifying and verifying sequential, concurrent and distributed systems. It is founded on two mathematical concepts:

i. the temporal logic of actions (TLA) for the characterization of the dynamic system behavior, and

ii. a variant of standard Zermelo-Fraenkel set theory with the axiom of choice (ZFC) for the description of data structures.

These mathematical aspects make TLA$^+$ easy to use and flexible to specify many different kinds of system. In addition, this mathematical approach allows the user to write a formal proof of a TLA$^+$ specification. While finding a proof is the most reliable way to reason about the correctness of a system, it is an extensive work and requires expert knowledge.

To automate the formal verification of TLA$^+$ specifications, a model checker, named TLC, was introduced. One main feature of TLC is that it explicitly enumerates reachable states. TLA$^+$ and TLC have been successfully used to specify and examine many industrial systems. However, TLC does not support any abstraction techniques used to reduce the complexity of a model. The lack of this feature seriously limits the applicability of TLC for large industrial systems. Thus, for the moment, TLC is seen as a tool whose main purpose is to detect simple bugs in TLA$^+$ designs.

Predicate abstraction is a prominent technique which made software model checking a reality. It reduces the state space of a large or infinite-state system. In this approach, an abstract model is automatically constructed from a given set of predicates over system variables and with the assistance of satisfiability modulo theories (SMT) solvers, which are used to evaluate the values of predicates and the transition relations.

In this thesis, we brought together TLA$^+$ with predicate abstraction technique guided by the user. We had to address three major challenges:

- TLA$^+$ is an untyped language, whereas SMT solvers are automatic theorem provers with decision procedures for several theories in many-sorted first-order logic.

- TLA$^+$ contains essential constructs that do not have counterparts in SMT-Lib, the language of SMT solvers.
- Transition systems are specified in TLA$^+$ as formulas with complex structures. For instance, a typical TLA$^+$ specification may contain about 100 lines of formulas in unsorted first-order logic with set constructs, involving quantification over records or tuples, constructs for functions and the CASE operator.

This thesis aims at developing a new model checker for TLA$^+$ specifications which can construct and verify an abstract model from a given set of predicates. In doing so, it makes two main contributions:

i. This thesis represents a sound translation from a fragment of TLA$^+$ to many sorted first-order logic. During the translation, constructs that do not have counterparts in SMT-Lib are eliminated by rewriting rules. Additional heuristics are applied to make the translation more simple and efficient. While our fragment introduces restrictions on TLA$^+$ expressions, we believe that it is still expressive and flexible enough to specify a large class of distributed algorithms. Our translation extends work by Merz and Vanzetto.

ii. This thesis develops a prototype model checker which supports predicate abstraction and the recent IC3 model checking technique. Instead of unrolling the transition relations, this model checker generates an incremental proof for a desired safety property. Our preliminary experiments show that a checker based on IC3 outperforms one with NuSMV2. The main disadvantage of the IC3 algorithm is that it cannot check liveness properties.

Keywords: TLA$^+$, safety, model checking, predicate abstraction, theorem proving, NuSMV2, IC3.

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Motivation

A distributed system is a collection of individual computing devices that can communicate with each other [AW04]. Nowadays, distributed computing plays an important role in many critical applications such as VLSI chips, multi-core systems, and safety-critical systems. Moreover, the development of methodologies and tools for specifying and reasoning about distributed algorithms has been an active area of research for the last few decades.

However, distributed algorithms are still difficult to design, build and verify. Their operation is very complex because their dynamic nature and features such as limited local knowledge, asynchrony and failures. These characteristics make distributed algorithms non-deterministic and construct an exponential number of execution scenarios [AW04]. In addition, computing models are traditionally described in natural language, their algorithms are still presented as pseudo-code, and their properties are proved manually [KVW12]. These reasons make it difficult to reason about distributed algorithms, even for experienced users. Therefore, errors and bugs in distributed systems are inevitable and lead to the loss of crucial data and unacceptable service outages. In spite of these challenges, the benefits and applications of distributed systems are abundant, making these systems worthwhile to chase.

In order to avoid ambiguity, prove the correctness and implement reliable systems, formal methods have been applied to model, specify and verify distributed systems and algorithms. First, formal methods are used to model and specify systems. Many frameworks and languagues such as input/output (I/O) automata, Event-B or TLA$^+$ have been suggested [LT87, Abr10, Lam02]. Their underlying mathematical aspects can be automata, set theory, classical logic or temporal logic. While most of the applications of Event-B appear to be control systems, I/O automata and TLA$^+$ concentrate on distributed systems [New14]. Second, formal methods are used to verify systems. The main reason of the need of formal verfication is that testing and simulation which usually involve providing certain inputs and observing the corresponding outputs can at best show the presence of errors but never their absence [BR70]. Since the 1970s, temporal logic has been used to prove properties of (concurrent) computer programs [Pnu77]. Since proofs were constructed by hand, the techniques introduced back then were often difficult to use in practice. While the introduce of interactive or automated theorem provers has made the construction of proofs faster, proving has still been intensive and burdensome work, requiring a large deal of human ingenuity. In the early 1980s, the appearance of temporal-logic model checking algorithms made this type of reasoning automated. This technique uses an exhaustive search of the state space of the system to determine if some property is true or not [CGP99]. Examples of model checkers include Spin [Hol97], TLC [LY01] and NuSMV2 [CCG$^+$02].

TLA$^+$, and its supporting tools, is one of the most prominent techniques to specify and reason about distributed systems. TLA$^+$ is a general-purpose formal specification language designed by Leslie Lamport for specifying and reasoning about concurrent and distributed algorithms and systems [Lam02]. The logical foundation of TLA$^+$ is a combination of the temporal logic of actions (TLA) [Lam94] and a variant of standard Zermelo-Fraenkel set theory with the axiom of choice (ZFC). The former is for the characterization of the dynamic system behavior and the latter is for the description of data structures. These characteristics make TLA$^+$ quick to learn, expressive to capture rich concepts and flexible to adjust the level of abstraction [New14]. Moreover, research community have developed powerful tools to construct a formal verification for TLA$^+$ specifications. These tools are the theorem prover TLAPS and the model checker TLC.

TLAPS. The TLA$^+$ Proof System, called TLAPS, is an interactive proof environment in which the user can deductively verify properties of TLA$^+$ specifications [CDLM08]. TLAPS is built around an application called the Proof Manager. The manager first interprets a TLA$^+$ proof as a collection of proof obligations. Then, the manager sends proof obligations to back-end theorem provers to prove. Finally, if possible, a verifier checks the proof generated by the back-end prover to provide complete machine-checking of TLA+ proofs. Back-end provers and verifiers used by TLAPS are Zenon, Isabelle/TLA$^+$ and SMT solvers, such as CVC, Z3 and VeriT. Theorem proving is a powerful method and writing proofs is a great reliable method to reason about properties of the system. However, constructing a proof is a complicated task and requires an assistance of an experienced user.

TLC. TLC is an model checker for TLA$^+$ specifications. In general, TLC checks the specification by exploring all reachable states and looking for one in which a desired property is not satisfied or deadlock occurs. If the property is violated, TLC will show a minimal length trace that leads from an initial state to the bad state. TLC stops when it has examined all reachable states in possible behavior. The main advantage of model-checking is that it requires much less effort and less expertise in the verification domain from the user than writing a fully formal proof of correctness. Typically, the user needs to add constraints to make the state space, run the tool, and wait for results. Therefore, TLC is used not only to verify a TLA$^+$ specification but also to check ideas occurring during the construction of a formal proof.

Shortcomings of TLAPS. While proving is the most reliable method to show the correctness of a system, it is intensive and burdensome work, requiring a large deal of human ingenuity. First, a proof usually starts with reasoning about an inductive invariant, but finding an inductive invariant is a difficult and error-prone task. Second, TLAPS does not offer any reasoning mechanism about many TLA$^+$ features such as recursive operators, real numbers and quantification over tuples and set constructors using tuples. In addition, the back-end provers require a lot of guidance to verify the CASE constructs, the CHOOSE operator, tuples, records… [1] Finally, a full formal proof may contain thousands of lines. For example, the formal specification and proof of Memoir contain 61 top-level definitions, 182 LET-IN definitions, 74 named theorems, and 5816 discrete proof steps [DLP$^+$11]. For the above reasons, it is difficult to use TLAPS in some cases. For example, Newcombe et al. tried to verify their critical algorithms by writing formal proofs but finally, he said "we doubt that we would use incremental formal proof as a design technique even for those algorithms" [New14].

Shortcomings of TLC The major challenge of model checking is the state-space explosion problem. To overcome it, the user usually needs to construct suitable abstractions of the system, which can reduce the number of states. However, TLC is still an explicit-state tool which explicitly enumerates reachable states and keeps all data on disk. That is, TLC does not support automated abstraction techniques. The lack of this feature limits seriously the applicability of TLC, especially to infinite-state systems.

Predicate abstraction Predicate abstraction is one of the most prominent methods to reduce the number of states and to construct an abstract model for a system with potentially infinite-state space [GS97]. The abstract model is constructed from a given set of predicates over program variables and with the help of SMT solvers to evaluate the values of predicates. In the early of 2000s, the introduce of the Counterexample-Guided Abstraction Refinement (CEGAR) technique allowed the refinement to be automated [CGJ$^+$00]. This method begins checking with a coarse (imprecise) abstraction and iteratively refines it. During the verification procedure, abstract models may admit erroneous (or "spurious") counterexamples. When a violation (counterexample) is found, the tool analyses it for feasibility. If the violation is feasible, it is reported to the user.

[1]We are not going to address all of these problems. At the moment, our system can reason about quantification over tuples and records, set constructs using tuples, constructs for tuples and records and the CASE operator. Automatic reasoning about recursive operators, real numbers, the CHOOSE operator... is still difficult problems for us.

If not, the proof of infeasibility is used to refine the abstraction automatically and the checking begins again with a new set of predicates. Until now, predicate abstraction with CEGAR has been used widely in many academic and industrial realistic projects.

1.2 Problem Statement

The main goal of this thesis is to develop a new prototype model checker for TLA$^+$ specifications which can construct and verify an abstract model from a given set of predicates which are manually specified by the user. In doing so, we need to

1. define a method to evaluate every states in a TLA$^+$ specification by given predicates,
2. find an efficient way to construct an abstract model,
3. and choose model checking algorithms to verify a reduced model.

1.3 Challenges

Applying predicate abstraction to the verification of a TLA$^+$ specification raise following challenges:

- TLA$^+$ is based on a variant of ZFC and therefore, it is an untyped language and all its well-formed expressions are sets. Moreover, there is no syntactic difference between Boolean and non-Boolean formulas in TLA$^+$. In contrast, SMT solvers are automatic theorem provers with decision procedures for several theories in many-sorted first-order logic (MS-FOL). Therefore, we need to define a type system for a fragment of TLA$^+$ and to establish a procedure for assigning appropriate type information to every expression in this fragments.
- Many constructs and expressions in TLA$^+$ do not have counterparts in many-sorted first-order logic. For example, we cannot translate constructs for functions in TLA$^+$ directly to MS-FOL. Another example is that a formula with quantifiers on predicate P in the set construct $\{x \in S : P\}$ is in second-order logic. As a result, in addition to restrictions on the fragment of TLA$^+$, we need to define a sound translation according with rewriting rules to eliminate "complex" expressions.
- Transition systems are specified in TLA$^+$ as formulas with complex structures. For instance, a typical TLA$^+$ specification may contain about 100 lines of formulas in unsorted first-order logic with set constructs, involving quantification over records or tuples, constructs for functions and the CASE operator. Therefore, absolutely unrolling the relation transition spends a tremendous amount of time and is inappropriate for many cases.

1.4 Related Works

Over the past years, there have been efforts to develop new model checkers for TLA$^+$ specifications. Hansen and Leuschel introduce a framework to translate TLA$^+$ to B for validation with ProB [HL12, LB03]. Later, Plagge and Leuschel integrate the Kodkod high-level interface to SAT-solvers into the kernel of ProB [PL12]. However, ProB is an explicit model checking tool like TLC and constructing a predicate abstraction for a B specification is not our focus.

In addition, researchers have made attempts on the integration of new automated or interactive theorem provers into TLAPS. This integration usually requires to translate TLA$^+$ expressions from the untyped non-temporal TLA$^+$ part into many-sorted (first-order) logic. Merz and Vanzetto represent approaches to encode TLA$^+$ proof obligations into many-sorted logic and to integrate automated theorem provers and SMT solvers into TLAPS. Each of their approaches can handle a different fragment of TLA$^+$.

First, Merz and Vanzetto try to encode sets by characteristic predicates but this method is so limited since it cannot represent a set of sets [MV12]. Later, the single-sorted encoding, which is also called the untyped encoding, for TLA$^+$ is suggested. This approach can handle a useful fragment of the TLA$^+$ language, including set theory, functions, linear arithmetic expressions and especially the CHOOSE operator (Hilbert's choice) [MV12]. The main weakness of the untyped encoding is that this mechanism introduces many additional quantifiers and defines many "fresh" relations, even for built-in operators in SMT-LIB.

In order to reduce the number of quantifiers and to utilize features in SMT solvers, Merz and Vanzetto continually propose a TLA$^+$ type system using refinement and dependent types [MV14]. However, this method is undecidable and if their typed system cannot decide an appropriate type for an expression, such as the empty set, they will come back to the untyped encoding. If we translate transition relations in a TLA$^+$ specification with their systems, it is extremely difficult for SMT solvers to reason about the resulting formulas automatically. The main reason is that a proof obligation is usually more "shallow" than a transition relation.

1.5 Contributions

Translation from TLA$^+$ to SMT-Lib The first contribution of this thesis is to define a type system and a type assignment procedure for a fragment of TLA$^+$. While this fragment does not support features such as recursive functions and the operator CHOOSE , it is a reasonable fragment which we believe can express many distributed algorithms. Its type system can be described in the language of state-of-the-art SMT solvers.

A model checker based on IC3 The second contribution of this thesis is to develop an model checker which is based the IC3 model checking algorithm and can construct an abstract model from a given set of predicates. Instead of absolutely unrolling the transition relation, this tool tries to construct an incremental inductive proof for a safety property in a system.

A model checker based on NuSMV2 The minor contribution of this thesis is to develop a model checker based on NuSMV2 which allows the user to verify safety properties of a TLA$^+$ specification by manually specifying predicates. This prototype was used in our preliminary experiments to compare model checking strategies.

1.6 Overview

The rest of this thesis is organized as follows:

- Chapter 2 gives a brief introduction to unsorted rewriting systems, first-order logic, SMT solvers and temporal logic.
- Chapter 3 presents the foundation of TLA$^+$, the structure of a TLA$^+$ specification, a theorem prover TLAPS and a model checker TLC.
- Chapter 4 describes how to verify a program with model checking algorithms, how to cope with the state-space explosion problem with predicate abstraction and how to find an inductive invariant with IC3.
- Chapter 5 is the main part of this thesis. It explains how to soundly translate a TLA$^+$ specification into the language of SMT solver and to eliminate complex expressions by rewriting rules. Moreover, it describes some heuristics to make reasoning about obtained formulas faster. Our translation is based on the work of Merz and Vanzetto [Van14].
- Chapter 6 depicts the architecture of our system and mentions additional elements in the structure of a TLA$^+$ specification. These elements are used to inform our model checker what predicates are used and what property needs checking.
- Chapter 7 discusses our experiments and results.
- Chapter 8 concludes and gives directions for future work.

CHAPTER 2

Preliminaries

In this chapter, we first introduce the foundation of rewriting systems. Second, we review the background of first-order logic, including both unsorted and many-sorted ones. Then, we summarize some essential concepts of satisfiability modulo theories (SMT), including the main formalisms in which the problems are expressed, the techniques which solvers implement, and the standard input and output format in the SMT community. Finally, we describe state-of-the-art SMT solvers.

As a form of computer program, rewriting systems was introduced in the late of 1960s [Gor65]. Nowadays, these systems play an important role important role in various areas, such as abstract data type specifications, implementations of functional programming languages and automated deduction. In our work, term rewriting rules are used to remove complex constructs in TLA$^+$.

An SMT problem generalises a Boolean satisfiability (SAT) problem by adding background theories such as: arithmetic, bit-vectors, arrays, and uninterpreted functions which are expressed in many-sorted first-order logic (MS-FOL) with equality. An SMT solver is a tool for deciding the satisfiability (or dually the validity) of an SMT problem. Our system uses a SMT solver to construct and reason about a "reduced" model.

Temporal logic has been applied successfully to specify and reason about concurrent and distributed algorithms. While now there are many variants of temporal logic, here we introduce only linear-time logic (LTL) since TLA$^+$ is based on LTL.

Chapter overview Section 2.1 introduces the foundation of (term) rewriting systems. Section 2.2 describes unsorted first-order logic and MS-FOL (MS-FOL) is presented in Section 2.3. Section 2.4 gives us some insights of their internal workings of SMT solvers. Section 2.5 describes the input formats SMT-LIB2 for SMT solvers and mentions some common background theories. Then, Section 2.6 outlines SMT competitions and explains why Z3, an SMT solver from Microsoft Research, is chosen as our back-end solver. Finally, Section 2.7 makes a brief introduction of LTL.

2.1 Rewriting system

The idea of simplifying expressions has been around as long as algebra has. The simplification procedure is usually to repeatedly replace subterms of a given expression with equal terms until the (possible) simplest form is obtained. As a form of computer program, rewriting systems was introduced in the late of 1960s [Gor65]. Nowadays, these systems play an important role important role in many areas, such as abstract data type specifications, implementations of functional programming languages and automated deduction. In addition to simplify expressions, our rewriting rules are used to remove complex constructs in TLA$^+$ which do not have counterparts in TLA$^+$, such as set constructs and operators.

We here present basic concepts of term rewriting systems (TRSs) and how to construct a terminate and confluent TRS from an equational theory. Several following definitions and lemmas are taken from Baader and Nipkow's Term Rewriting and All That [BN98].

An *abstract rewriting system* (ARS) models step by step activities like a transformation of some object (e.g., a term) or stepwise execution of computations. Formally, an ARS contains a set of objects A and a binary relation $\rightarrow$ on A, i.e. $\rightarrow \subseteq A \times A$, called *rewriting* or *reduction relation*. We usually denote an ARS as $\langle A, \rightarrow \rangle$. When $\langle a, b\rangle \in \ \rightarrow$ for $a, b \in A$, we simply write $a \rightarrow b$ and say that there is a *step* from a to b.

We define the following symbols by composing a relation with itself:

$\xrightarrow{0} ::= \{(x,x) \mid x \in A\}$	identity
$\xrightarrow{i+1} ::= \xrightarrow{i} \circ \rightarrow$	$(i+1)$-fold composition, $i \geq 0$
$\xrightarrow{+} ::= \bigcup_{i>0} \xrightarrow{i}$	transitive closure
$\xrightarrow{*} ::= \xrightarrow{+} \cup \xrightarrow{0}$	reflexive transitive closure
$\xrightarrow{=} ::= \rightarrow \cup \xrightarrow{0}$	reflexive closure
$\xrightarrow{-1} ::= \{(y,x) \mid x \Rightarrow y\}$	inverse
$\leftarrow ::= \xrightarrow{-1}$	inverse
$\leftrightarrow ::= \rightarrow \cup \leftarrow$	symmetric closure
$\overset{+}{\leftrightarrow} ::= (\leftrightarrow)^+$	transitive symmetric closure
$\overset{*}{\leftrightarrow} ::= (\leftrightarrow)^*$	reflexive transitive symmetric closure

Based on these symbols, we introduce some extra notions:

1. x is *reducible* if there is a y such that $x \rightarrow y$.

2. x is *in normal form* (irreducible) if it is not reducible.

3. y is *a normal form* of x if $x \overset{*}{\rightarrow} y$ and y is in normal form. If x has a uniquely determined normal form, the latter is denoted by $x\downarrow$.
4. y is *a direct successor* of x if $x \rightarrow y$.
5. y is *a successor* of x if $x \overset{+}{\rightarrow} y$.
6. x and y are *joinable* if there is a z such that $x \overset{*}{\rightarrow} z \overset{*}{\leftarrow} y$, which is also written as $x \downarrow y$.

A reduction relation $\rightarrow$ is called

- *Church-Rosser* if $x \overset{*}{\leftrightarrow} y$ then $x \downarrow y$,
- *confluent* if $y_1 \overset{*}{\leftarrow} x \overset{*}{\rightarrow} y_2$ then $y_1 \downarrow y_2$,
- *terminating* if there is no infinite rewriting sequence $a_0 \rightarrow a_1 \rightarrow \ldots$,
- *normalizing* if every term has a normal form,
- *convergent* if it is both confluent and terminating, and
- *locally confluent* if $y_1 \leftarrow x \rightarrow y_2$ then $y_1 \downarrow y_2$.

Now we consider rewriting systems whose objects are terms [1], usually called *term rewriting system* (TRS). An equation $s \approx t$ is a pair $\langle s, t\rangle \in T \times T$, where T is the set of terms. A TRS is used to check for equality in an equational theory by rewriting both sides of the identity in question. Equations, especially its laws, in a given equational theory can be used to generate rewriting rules. An equation $s \approx t$ where s is not a variable and t contains only variables appearing in s is called a rewrite rule and is usually written $s \longrightarrow t$. Identities can transform terms into other "equivalent" terms by replacing instances of the left-hand side (lhs) with the corresponding instances of the right-hand side (rhs), and vice-versa. A *redex* (reducible expression) is an instance of the lhs of a rewrite rule. Rewriting or reducing the redex means replacing it with the corresponding instance of the rhs of the rule.

Finally, we can define the concept of critical pair, which is crucial for proving TRS confluence and termination. Let $a|_p$ be a sub-term of a at a given position p in its syntactic tree and $a[b]_p$ be the term a such that b replaces $a|_p$. Consider two rules $a_1 \longrightarrow b_1$ and $a_2 \longrightarrow b_2$ whose variables have been renamed such that these rules do not share variables. Let p be a position in the syntactic tree of a_1 such that $a_1|_p$ is not a variable, and σ be a most general unifier (mgu) of $a_1|_p$ and a_2, that is, the superposition of the left-hand sides of both rewriting rules. Then, the pair $\langle b_1\sigma, (a_1[b_2]_p)\,\sigma\rangle$ is *a critical pair.*

Now, we have three lemmas about the TRS confluence and termination.

[1] For the definition of terms, please read the section 2.2

Lemma 2.1. A terminating ARS is confluent if it is locally confluent.

Lemma 2.2. A TRS is locally confluent if and only if all its critical pairs are joinable.

Lemma 2.3. A terminating TRS is confluent if and only if all its critical pairs are joinable.

Completion and orderings An interesting question is how to construct a terminating and confluent TRS from a finite set of identities (or equations). Knuth and Bendix presented a semi-decision algorithm to solve this problem in [KB83]. Provided a well-founded ordering $\succ$ on terms, their procedure takes a set E of equations $s \approx t$ between terms and applies the inference rules $\Longrightarrow_{KB}$ to construct a terminating and confluent TRS. Initially, the inference rules are applied to the pair $E, \varnothing$, where the second element represents a set of rewriting rules which a generated TRS contains. Two main inference rules are rules (Orient) and (Deduce). Rule (Orient) transforms an equation $s \approx t$, if $s \succ t$, into the rewriting rule $s \longrightarrow t$. Rule (Deduce) detects a critical pairs of $\langle s, t\rangle$ and add a new equation $s \approx t$ to E. During the process, some equations are simplified, and trivial ones such as $s \approx s$, which cannot be directed, are removed. The procedure succeeds if it generates a final pair $\varnothing, R$, where the final set R is a set of rewriting rules equivalent to E. The main weakness of this algorithm is that it may terminate with failure or run forever if there exists an equation which can neither be oriented or removed. In order to ensures termination, the ordering is total on ground terms [BDP89]. For instance, assuming some basic requirements, the Knuth-Bendix ordering $\succ_{KB}$ which uses a lexicographic comparison of terms together with a weight function is a total ordering on ground terms. An algorithm using this kind of ordering is called an *unfailing* one.

2.2 Unsorted first-order logic

In this section we describe the syntax and semantics of (unsorted) first-order logic (FOL) and conjunctive normal form (CNF) in first-order and propositional logics.

The language FOL is the classical (unsorted) first-order logic with equality.

2.2.1 FOL syntax

We assume given three non-empty, infinite and disjoint collections:

- $\mathcal{V}$ of *variable* symbols,
- $\mathcal{F}$ of *function* symbols [2], and

[2]CNF, FOL and MS-FOL functions should not to be confused with TLA+ functions

- $\mathcal{P}$ of *predicate* symbols.

In addition, we assume the total function

- $ar : \mathcal{F} \cup \mathcal{P} \to \mathbb{N}$

that assigns a natural number, the arity, to each symbol in $\mathcal{F}$ and $\mathcal{P}$. Together, they define the FOL *signature* $\Sigma = \langle \mathcal{V}, \mathcal{F}, \mathcal{P}, ar \rangle$ that fixes an alphabet of non-logical symbols. We call the nullary symbols of $\mathcal{F}$ *constant* symbols and usually denote them by the letters a, b possibly with subscripts. We call the nullary symbols of $\mathcal{P}$ *propositional* symbols, and usually denote them by the letters A, B, possibly with subscripts. Also, we use f, g, possibly with subscripts, to denote the non-constant symbols of $\mathcal{F}$, and p, q, possibly with subscripts, to denote the non-propositional symbols of $\mathcal{P}$.

FOL has two syntactical categories: *terms* t and *formulas* φ.

$$
\begin{aligned}
t &::= x \mid f(t_1, \ldots, t_n) \\
\varphi &::= \bot \mid \varphi \Rightarrow \varphi \mid \forall x.\varphi \mid t = t \mid p(t_1, \ldots, t_m)
\end{aligned}
$$

A term t is either a variable symbol x in $\mathcal{V}$, or an application $f(t_1, \ldots, t_n)$ of a function symbol f (with $ar(f) = n$) in $\mathcal{F}$ to n terms $t_1, \ldots, t_n$. A *ground term* is a term without any variable. A formula φ is either the falsehood symbol $\bot$, an implication between two formulas, a universal quantification of a formula, an equality between terms, or an application $p(t_1, \ldots, t_m)$ of a predicate symbol p (with $ar(p) = m$) in $\mathcal{P}$ to m terms $t_1, \ldots, t_m$. The last two kind of formulas are called *atoms*. A formula without variables is called a *ground formula*.

Additionally, we define the familiar constant $\top$ (truth), the connectives $\neg, \wedge, \vee, \Leftrightarrow$, and the existential quantifier $\exists$. In particular, universal and existential quantifiers can be freely nested. The definitions of *free variables* and *variable substitution* are the standard ones [EE01]. By $FV(\varphi)$, we note the free variables of a formula φ. By $\varphi_1[x \leftarrow \varphi_2]$, we denote the formula φ_1 where the free variable x is substituted by the formula φ_2. By l, we denote a *literal* which is either an atom (a positive literal) or the negation of an atom (a negative literal).

2.2.2 FOL semantics

A FOL *model* (also called structure) $M = \langle \mathcal{D}, v, \mathcal{I} \rangle$ is composed of

- a non-empty (infinite) set $\mathcal{D}$ called the *domain* or *universe of discourse*,
- a *valuation function* $v : \mathcal{V} \to \mathcal{D}$ that assigns to each variable an element in the domain, and

- an *interpretation function* $\mathcal{I}$ that assigns to each function symbol f in $\mathcal{F}$ a function $\mathcal{I}(f) : \mathcal{D}^{ar(f)} \to \mathcal{D}$, and to each predicate symbol p in $\mathcal{P}$ a set $\mathcal{I}(p) \subseteq \mathcal{D}^{ar(p)}$.

The *interpretation* of first-order formulas is defined in the standard way [EE01]. The valuation of a term t under a model $M = \langle \mathcal{D}, v, \mathcal{I} \rangle$ noted $val_M(t)$, is inductively defined by:

$$\begin{aligned} val_M(x) &= v(x) \\ val_M(f(t_1, \ldots, t_n)) &= \mathcal{I}(f)(val_M(t_1), \ldots, val_M(t_n)) \end{aligned}$$

The truth value of a formula φ under a model $M = \langle \mathcal{D}, v, \mathcal{I} \rangle$, noted $M \vDash \varphi$ or $\mathcal{D}, v, \mathcal{I} \vDash \varphi$, is inductively defined as:

$$\begin{aligned} &M \nvDash \bot \\ &M \nvDash \varphi_1 \Rightarrow \varphi_2 \text{ iff } M \vDash \varphi_1 \text{ and } M \nvDash \varphi_2 \\ &M \vDash \forall x.\varphi \text{ iff } \mathcal{D}, v \oplus (x \mapsto d), \mathcal{I} \vDash \varphi \text{ for all } d \in \mathcal{D} \\ &M \vDash t_1 = t_2 \text{ iff } val_M(t_1) \text{ is equal to } val_M(t_2) \\ &M \vDash p(t_1, \ldots, t_n) \text{ iff } \langle val_M(t_1), \ldots, val_M(t_n) \text{ is a member of } \mathcal{I}(p) \end{aligned}$$

where $v \oplus (x \mapsto d)$ denotes a valuation that is equal to v for all variables in $\mathcal{V}$ except in x and mapping x to d. We say that a model M satisfies a formula φ if and only if φ evaluates to true under M. A formula φ is called *satisfiable* iff there exists a model M such that M satisfies φ. Otherwise, φ is called *unsatisfiable*. A formula φ is *valid*, noted $\vDash \varphi$, iff $M \vDash \varphi$ for all models M, that is, for every domain, valuation and interpretation.

2.2.3 Conjunctive Normal Forms

A *clause* C is a set of literals $l_1, \ldots, l_n$ that represents the disjunction of its elements. In other words, we have that $C = l_1 \vee \ldots \vee l_n$. And the empty clause $\bot$ is interpreted as falsehood. A clause with one literal is called a unit clause. A formula φ is in *conjunctive normal form* (CNF) if and only if it is a set of clauses $C_1, \ldots, C_n$, where variables are interpreted universally, that is, no variables are bound. A *propositional CNF* formula is a ground first-order CNF formula.

Given a propositional formula φ, the Boolean satisfiability (SAT) problem is to decide whether there exists a model M such that $M \vDash \varphi$. Nowadays, SAT solvers usually require that the input formula φ is in CNF. Moreover, if we want to check whether a formula φ is valid, we just prove that $\neg\varphi$ is unsatisfiable or find a derivation of the empty clause from the given clauses using some appropriate mechanics such as natural deduction or sequent calculus.

2.3 Many-sorted first-order logic

In many practical applications, it is desirable to categorize objects in many different types, or sorts. For instance, common data structures found in computer science such as integers, reals, and strings are usually formalized in different categories. By adding to the formalism of first-order logic the notion of type, one obtains a flexible logic called many-sorted first-order logic, which has similar properties as first-order logic. While sorts require some restrictions on formulas [Coh87], they bring momentous advantages to automated theorem provers: the type discipline implicitly avoids pointless inferences by not having to evaluate ill-sorted formulas. In other words, sorts cut the search space by eliminating useless branches.

The MS-FOL language is an extension of FOL with an extra syntactical category of sorts, i.e. basic types. The set $\mathcal{V}$ of variable symbols is divided by sorts, while functions and predicates range over sorts as well.

2.3.1 MS-FOL syntax

We assume four given non-empty enumerable infinite, and disjoint collections

- $\mathcal{S}$ of *atomic sort* symbols,
- $\mathcal{V} = \bigcup_{\sigma \in \mathcal{S}} \mathcal{V}_\sigma$, the (enumerable) union of sets $\mathcal{V}_\sigma$ of *variable* symbols of sort σ,
- $\mathcal{F}$ of *function* symbols, and
- $\mathcal{P}$ of *predicate* symbols.

In addition, we assume given the functions

- $ar : \mathcal{F} \cup \mathcal{P} \Rightarrow \mathbb{N}$ which assigns an *arity* to each symbol in $\mathcal{F}$ and $\mathcal{P}$, and
- $\Theta : \mathcal{F} \cup \mathcal{P} \Rightarrow \mathcal{S}^*$ which assigns to functions f in $\mathcal{F}$ a value in $\mathcal{S}^{ar(f)+1}$ (the Cartesian product with $ar(f) + 1$ dimensions), and to predicates p in $\mathcal{P}$ a value in $\mathcal{S}^{ar(p)}$.

Together, they define an MS-FOL signature $\Sigma = \langle \mathcal{S}, \mathcal{V}, \mathcal{F}, \mathcal{P}, ar, \Theta \rangle$ [3].

The language MS-FOL has three syntactical categories: sorts, (well-sorted) terms, and formulas. A sort σ is just an atomic sort symbol in $\mathcal{S}$. Note that the values $\Theta(f)$ and

[3]The arity of a function or predicate symbol f can be got back from $\Theta(f)$ but we keep it in the signature to maintain MS-FOL as an extension of FOL.

$\Theta(p)$ associated to a function symbol f and a predicate symbol p, when $ar(p) > 1$, are no sorts.

$$\begin{aligned} t &::= x \mid f\,(t_1, \ldots, t_n) \\ \varphi &::= \bot \mid \varphi \Rightarrow \varphi \mid \forall x^{\sigma}.\varphi \mid t = t \mid p\,(t_1, \ldots, t_m) \end{aligned}$$

A term t of sort σ is either a variable symbol x in $\mathcal{V}_{\sigma}$, or a sort-consistent application of a sorted function symbol f with $\Theta(f) = \langle \sigma_1, \ldots, \sigma_n, \sigma \rangle$ in $\mathcal{F}$ to n terms $t_1, \ldots, t_n$ of sort $\sigma_1, \ldots, \sigma_n$, respectively. A formula σ is built as in unsorted FOL except that

- each quantified variable is annotated with a sort,
- equality is ad-hoc polymorphic over the sorts, i.e. its arguments must have the same but arbitrary sort, and
- predicates applied to terms must be sort-consistent, that is, a predicate symbol p in $\mathcal{P}$ with $\Theta(p) = \langle \sigma_1, \ldots, \sigma_m \rangle$ should be applied to m terms $t_1, \ldots, t_m$ of sort $\sigma_1, \ldots, \sigma_m$, respectively.

2.3.2 MS-FOL semantics

The semantics of MS-FOL is analogous to FOL with the adjustments corresponding to the presence of sorts. Moreover, the sort machinery is in principle not needed because sorts can be encoded using unary predicates by relativizing quantified sorted variables. Relativization is the traditional method to translate a multi-sorted language into a single-sorted one [EE01]. For every atomic sort $\sigma \in \mathcal{S}$, a fresh unary characteristic predicate P_{σ} that represents the set of values having that sort is introduced. For each variable x with sort σ, one new formula $P_{\sigma}(x)$ is added. A MS-FOL formula is relativized by systematically replacing the quantified sorted formulas $\forall x_{\sigma}.\varphi$ by a FOL formula $\forall x.P_{\sigma}(x) \Rightarrow \varphi$. Moreover, these predicates partition the universe of atomic sorts in disjoint sets. For each pair of sorts $\langle \sigma_1, \sigma_2 \rangle$, the encoding introduces an extra axiom

$$\forall x, y.P_{\sigma_1}(x) \land P_{\sigma_2}(y) \Rightarrow \neg(x = y)$$

Lemma 2.4. (Relativization is sound [EE01]). $\vDash \forall x^{\sigma}.\varphi$ implies $\vDash \forall x.P_{\sigma}(x) \Rightarrow \varphi$.

Consequently, all the main results of the unsorted logic extend to the many-sorted case. Moreover, if the set $\mathcal{S}$ contains only one sort, the logic MS-FOL becomes single-sorted, thus equivalent to FOL [EE01].

The traditional MS-FOL requires that each sort is interpreted as a set not overlapping that of any other sorts, i.e. sorts are interpreted as disjoint sets. Some variants relax this restriction to be more expressive and a sort structure can be represented as a tree or a lattice. However, the overlapping between sorts makes the inference procedure more complex. Therefore, in the rest of our work, we consider only the traditional sort structure.

2.4 SMT Solvers

In order to give the reader an idea of the kind of formulas that SMT solvers can handle, we informally give a compact description of their components and internal workings. In addition, we first overview some basic concepts of rewriting systems and simplification techniques, used by SMT solvers in the preprocessing steps and applied later in the following chapter.

2.4.1 Simplification

Preprocessing is crucial for efficient reasoning and modern first-order theorem provers spend a great part of their time in simplification operations. Most operations are inferences that remove or modify existing formulas. In the case that the domain is finite, universal quantifiers can be deleted.

1. **Miniscoping**: Miniscoping is a common technique for minimizing the scope of quantifiers [17]. We can apply it after transform a formula to negation normal form (NNF), that is the negation operator is only applied to variables and a formula accepts only two other Boolean operators which are conjunction and disjunction. The main idea is to distribute universal (existential) quantifiers over conjunctions (disjunctions). We may also limit the scope of a quantifier if a sub-formula does not contain the bound variable. It means

   $$(\forall x.F[x] \vee G) \longrightarrow (\forall x.F[x]) \vee G$$

   when G does not contain x. We use a similar rule for existential quantifiers over disjunctions.

2. **Skolemization**: Existentially quantified variables can be eliminated by using Skolemization which removes existential quantifiers by introducing new function symbols. This procedure keeps the equisatisfiability of a formula, not the equality. For example, a formula $\forall x.\exists y.p(x, y)$ is transformed into the equisatisfiable formula $\forall x.px, f_y(x))$, where f_y is a fresh function symbol.

3. **Rewriting**: We can use rewriting rules to simplify our formula.

4. **Other simplifications**: If we know the type of variable x, we can apply other rules. For example, we have $a + 0 \longrightarrow a$ or $a - a \longrightarrow 0$ if a is an integer.

2.4.2 SAT solving

The Boolean satisfiability (SAT) problem is the problem of determining whether there exists a interpretation that satisfies a given Boolean formula [DP60, DLL62]. The DPLL procedure is a complete, backtracking-based algorithm for checking the satisfiability of propositional logic formulas in CNF. Until now, DPLL is still the basis procedure for most efficient complete SAT solvers, as well as for many theorem provers for fragments of first-order logic.

The basic backtracking algorithm works by choosing a literal, assigning a truth value to it, simplifying its clauses and then recursively checking whether the new (simplified) formula is satisfiable. If yes, the original formula is satisfiable. Otherwise, the algorithm assumes the opposite truth value and then applies the same recursive check again. This is considered as the splitting rule, since it splits the search space into two simpler and smaller parts. The simplification step essentially eliminates all clauses which become true under the assignment from the formula, and all literals that become false from the remaining clauses. Notice that the order in which the search space is explored heavily relies on the choice of the splitting variables.

The DPLL algorithm strengthens the backtracking algorithm by applying the following rules at each step:

- Unit propagation: If a clause is a unit clause, this clause can only be satisfied by assigning the appropriate value to make this literal true. It means that there is only one choice. In other words, if there exists a clause $C_j = q$, then we drop it, every clause containing q, every literal $\neg p$ from the remaining clauses. In practice, this help us avoide a large part of the naive search space.
- Pure literal elimination: If a propositional variable occurs only positively or negatively in the formula, it is called pure. Pure literals can always be assigned in a way that makes all clauses with them true. Thus, these clauses can be deleted. While this optimization is part of the original DPLL algorithm, this techniques are usually omitted in current SAT solvers, since the overhead for detecting purity, even negative.

Modern SAT solvers implement variants of the classical algorithm, which include techniques like back-jumping, conflict-driven learning, and restarts among others, plus many optimizations, such as the use of highly efficient data structures, and heuristics to select splitting variables [Kul09, MS99]. The performance of different SAT solvers also depend on how they implement the preprocessing methods. Although SAT solvers work extremely well in practice, checking the unsatisfiability of CNF formulas is coNP-complete.

2.4.3 SMT solving

We first explain how to utilize a SAT solver as an engineer in an SMT solver. A simple integration allows us to solve quantifier-free SMT problems. To work with quantified formulae, techniques for eliminating and instancing quantifiers need applying. These techniques are also mentioned in following sub-sections.

Quantifier-free SMT solvers

A theory T is a set of closed first-order formulas. The common first-order theory are the theory of equality and uninterpreted functions (EUF), fragments of arithmetic (linear or non-linear, integer or real), arrays and bit vectors. Function symbols of a formula φ occurring in a theory T are interpreted, while others are not in T are uninterpreted. A first-order formula φ is T-satisfiable if $T \cup \{\varphi\}$ is satisfiable, that is, if there exists a model of T, that is also a model of φ. Otherwise, φ is T-unsatisfiable. Each T-theory has a T-solver which is an efficient procedure for the satisfiability of conjunctions of literals in T.

Satisfiability modulo theories (SMT) solvers rely on an efficient propositional engine, typically a SAT solver, with specialized T-solvers. The common approach is as follows

1. reduce an SMT formula into a new simplified SAT formula,
2. use an SAT solver to check the satisfiability of a new formula and to generate a model, and
3. call a T-solver to verify an obtained model.

A SAT formula is usually created by abstraction which maps atoms of the original formula into fresh Boolean variables. The new formula is equisatisfiable and can be checked by a SAT solver. If the abstract formula is found to be unsatisfiable, then so, too, is the original SMT formula. On the other hand, if the SAT procedure finds the abstract formula to be satisfiable, the T-theory solver is called to check the model generated by the SAT solver. This interplay is called the “lazy” approach.

The above algorithm is for only one theory. In many areas, we need to combine multiple theory solvers. For example, software verification usually requires a combination of arithmetic and arrays solver. Principal questions include [dMB08]:

- Is the union of two decidable theories still decidable?
- Is the union consistent?
- And how can we combine different theory solvers?

In general, combining theory solvers is an extremely difficult problem. Fortunately, useful special cases have good results. The Nelson-Oppen architecture is a well-known approach for the combinations of decision procedures in different theories [NO79]. In this framework, theories must satisfy two conditions:

- they are disjoint, meaning that they share equality as the only interpreted symbol, and

- they are stably infinite.

A theory T is stably infinite if whenever a (quantifier-free) formula is T-satisfiable, then it is T-satisfiable in a model of the theory with an infinite universe. Fortunately, many interesting theories such as theories for integers, real numbers and arrays are disjoint and stably infinite. Therefore, we can combine them into one theory.

To solve an SMT problem, the theory procedures need to communicate by exchanging equalities of variables through the EUF solver.

Quantifier instantiation

Since the Nelson-Oppen framework only works for quantifier-free formulas, SMT solvers were rather limited in their ability to reason about quantifiers. Existential quantifiers are usually eliminated by Skolemization, but universal quantifiers are difficult to remove. The notable solution is quantifier instantiation which apply different heuristics to choose a set of ground instances $\{t_1, \ldots, t_n\}$ of the quantified sorted variable x. Instead of $\forall x^{\sigma}.\varphi$, the formulas $\varphi\,[x \leftarrow t_i]$, which are logical consequences of the original formula, are added to the clause set to continue the quantifier-free search.

Pattern-based instantiation, also known as E-matching, is a well-known strategy to find just the instances that render the problem unsatisfiable [DMB07]. This heuristic finds ground terms that have the same shape as sub-terms of the body of $\forall x^{\sigma}.\varphi$, and uses them to guide the instantiation. For example, suppose that we have a set of formulas:

1. $\forall x.f\,(g\,(y)\,, a) = a$
2. $g\,(b) = c$
3. $f\,(c, a) = b$
4. $\neg\,(b = c)$

Pattern-based instantiation finds in the quantified formula the pattern $g\,(y)$ that matches the ground terms $g\,(b)$ with the substitution $y \mapsto c$. As a result of applying the substitution in the quantifier, we obtain $f\,(g\,(b)\,, a)$, which equals $f\,(c, a)$ and b. It causes a conflict at the ground level $b = c, \neg\,(b = c)$. Although relatively useful for some applications, pattern-based instantiation is not refutationally complete, needs ground seeds, and could generate an exponential number of matches (most of them useless), which may generally decrease the solver's efficiency. Moreover, the syntactic structure of φ, which is particularly affected by equivalence-preserving transformations of the original formula in the preprocessing step, plays an important role. In order to reduce the number of instances generated, the user can guide the instantiation by providing triggers, which are sub-terms occurring in φ, used as hints of potentially useful patterns.

Alternatively, the model-based approach for complete quantifier instantiation (MBQI) [GDM09] uses model-checking techniques to prioritize and to recognize model candidates, without explicitly generating all instances. MBQI is effective to construct models of satisfiable formulas and, and in practice, it sometimes outperforms pattern-based in solving unsatisfiable cases as well. For some fragments of first-order logic, the MBQI engine guarantees that the SMT solvers are decision procedures.

In conclusion, the quantifier instantiation procedure is incomplete, since no fair strategy guarantees the generation of the ground instances that are necessary to derive the empty clause. Until now, these problems with nested quantifiers are very difficult to solve. In these cases, the SMT solver either runs indefinitely with useless ground clauses or reports "unknown" together with a partial model satisfying only the grounded instances.

2.5 The satisfiability modulo theories library

The SMT-LIB standard [SMT16] has the goal of advancing the theory and practice of SMT solvers by

- giving standard rigorous descriptions of background theories used in SMT systems,
- providing a repository of benchmarks to the SMT community, and
- developing a common input and output format for SMT solvers.

At the assertion level, an SMT-LIB file is a sequence of

- commands to configure and interact with the solvers,
- declarations and definitions of sort and function symbols, and
- assertions of formulas over the resulting signature.

An SMT-LIB problem is given as a list of assertions, where the conjecture is negated. SMT-LIB requires all identifiers to be declared before using them.

At the logical level, the SMT-LIB language is a variant on MS-FOL which allows some syntactic features of higher-order logics: in particular, the identification of formulas with terms of a distinguished Boolean sort, and the use of sort symbols of arity greater than 0. As a result, for example, it is possible to have a sort List(Array(Int, U)), where Int, U, List, and Array are sort constructors with arities 0, 0, 1 and 2, respectively. However, it still requires that

- quantifiers are still first-order,

- the sort structure is flat (no subsorts),
- the logic's type system has no function (arrow) types, and
- there are no type quantifiers, no dependent types, no provisions for parametric or subsort polymorphism.

Moreover, each legal expression must have a unique sort that has to respect the sorting discipline. It means these properties of traditional MS-FOL still remains in SMT-LIB.

In SMT-LIB, Bool is a predefined sort. In contrast to MS-FOL, a predicate is defined as a function that returns a Bool-sorted term, and SMT-LIB formulas are just terms of sort Bool, i.e. they are both included in the same syntactic category t. In addition, SMT-LIB provides a conditional if-then-else operator as a term, noted ite, where the sort of the first argument is Bool, and the second and third arguments have the same sort.

SMT-LIB defines clearly a catalog of background theories, and give an implicit description of combined theories by means of a general modular combination operator. An SMT logic is identified by a pre-established name to which are associated sort and function declarations, and possibly syntactic and semantic restrictions.

The background theories are QF_AX, QF_IDL, QF_UF, QF_BV and QF_RDL [SMT16]. Letter groups in these names evoke the theories used by the logics and some major restriction in their language, with the following conventions:

- QF for the restriction to quantifier free formulas,
- A or AX for the theory ArraysEx,
- BV for the theory FixedSizeBitVectors,
- FP (forthcoming) for the theory FloatingPoint,
- IA for the theory Ints (Integer Arithmetic),
- RA for the theory Reals (Real Arithmetic),
- IRA for the theory Reals_Ints (mixed Integer Real Arithmetic),
- IDL for Integer Difference Logic,
- RDL for Rational Difference Logic,
- L before IA, RA, or IRA for the linear fragment of those arithmetics,
- N before IA, RA, or IRA for the non-linear fragment of those arithmetics, and
- UF for the extension allowing free sort and function symbols.

We are mainly interested in the logic AUFLIA that offers a pre-defined sort Int, quantified formulas and arithmetic expressions, features that are ubiquitous in hardware and software verification problems [4]. In our TLA+ encodings of Chapter 5, we will focus only on the translations to the AUFLIA fragment.

2.6 SMT solver competitions and Z3

The annual Satisfiability Modulo Theories Competition (SMT-COMP) was initiated in 2005 in order to develop new theories and logics for SMT, and to encourage the advance of state-of-the-art techniques and tools developed by the SMT community [BDdM$^+$13]. The competition have two tracks: the “main” track and the “application” track. In both tracks, the main task of a solver is to determine the satisfiability or unsatisfiability of a single problem in a logical theory. While problems in the first track are usually very different, problems in the second one may have a substantially similar set of assertions. Hence, we can consider that in the second track, a solver needs to deal with incremental problems. Every solver is checked in the sequential and parallel modes. In the SMT-COMP 2015, 21 solvers competed in 41 logical divisions and these following tables show top-three solvers Z3, CVC4 and Yices which support all or nearly all logics and earn the highest scores in various divisions [SMT15]. Moreover, these solves have performed at the highest levels in previous competitions.

Sequential Performances		
Rank	Solver	Score
-	[Z3]	159.36
1	CVC4	144.67
2	CVC4 (exp)	140.47
3	Yices	101.91

Parallel Performances		
Rank	Solver	Score
-	[Z3]	159.36
1	CVC4	144.74
2	CVC4 (exp)	140.51
3	Yices	101.91

Sequential Performances (industrial benchmarks)		
Rank	Solver	Score
-	[Z3]	139.34
1	CVC4	124.59
2	CVC4 (exp)	120.49
3	Yices	81.64

Parallel Performances (industrial benchmarks)		
Rank	Solver	Score
-	[Z3]	139.34
1	CVC4	124.63
2	CVC4 (exp)	120.51
3	Yices	81.64

Because of these above reasons, Z3, an SMT solver from Microsoft Research [dMB08], is chosen as our back-end solver. It is targeted at solving problems that arise in software verification and software analysis. Z3 integrates a modern DPLL-based SAT solver, a core

[4] AUFLIA is a theory which contains closed formulas over the theory of linear integer arithmetic and arrays extended with free sort and function symbols but restricted to arrays with integer indices and values.

theory solver that handles equalities and uninterpreted functions, satellite solvers (for arithmetic, arrays, etc.) and techniques for quantifier instantiation. Z3 is implemented in C++ and supports APIs for many languages such as Python, C/C++, .NET and Java. At the language aspect, Z3 allows the SMT-LIB input format and supports extensions such as data-types, weight annotations and pattern definitions.

Z3 is being used in several realistic projects in academia and industry and its main applications are extended static checking, test case generation and predicate abstraction [BDM09]. In our thesis, Z3 is used as a backend prover to construct the predicate abstraction and to check safety properties.

2.7 Linear-time logic

Propositional logic and first-order logic are useful tools to describe statements whose truth values do not change from time to time. However, these logics are not convenient enough to formalize sentences whose truth values depend on time. To reason about time, both propositional logic and first-order logic require that points of time have to be explicitly represented in the underlying universe. Therefore, the user needs to construct very complicated formulas in these logics to represent successfully some interesting properties of concurrent program.

Temporal logic addresses this issue by containing some reference to time conditions in which the user can describe sequences of changes and properties of behavior [5]. This property makes temporal logic adequate for expressing a broad variety of behavior of concurrent systems such as termination (the program eventually does terminate) or starvation-freedom (a process eventually receives services). Therefore, temporal logic is applied to following topics: formal specification, formal verification and requirements description [Lam83, Pnu77]. At the moment, there exists many variants of temporal logic such as linear-time logic, branching-time logic, time-intervals logic or partial-order logic. However, in this section, we introduce only linear-time logic (LTL) where formulas are interpreted on one (linear) execution of the system. Temporal logic of Action (TLA), a well-known variant of LTL is introduced by Lamport, is discussed in Chapter 3

2.7.1 Syntax

LTL formulas are defined with the following grammar

$$\varphi ::= \top \mid p \mid \varphi \vee \varphi \mid \neg\varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$$

where

- p ranges over a (finite) set AP of atomic propositions, and

[5]Temporal logic is used to describe the order in which events must happen rather than the actual times at which they happen.

- **X** and **U** are also *temporal operators* called "next" and "until", respectively.

Other Boolean connectives in classical logic are defined in the standard way. We can define two extra operators "eventually" **F** and "always" **G** by following rules

$$\mathbf{F}\varphi = \top\mathbf{U}\varphi$$
$$\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$$

2.7.2 Kripke structures

Let AP be a set of atomic propositions. A *Kripke structure* is a tuple $K = (S, S_0, R, L)$, where

- S is a set of *states*,
- $S_0 \subseteq S$ is a set of *initial* states,
- R is the left-total *transition relation* on $S \times S$, i.e. for every state $s \in S$, there exists state $s' \in S$, such that $R(s, s')$, and
- L is a *labelling function* $L : S \to 2^{AP}$ which defines for each state $s \in S$ the set $L(s)$ of all atomic propositions that are valid in s.

Since R is left-total, it is always possible to construct an infinite path π through a Kripke structure. A *path* of the structure K is a sequence of states $\pi = s_0, s_1, \ldots$ such that for each $i \geq 0, R(s_i, s_{i+1})$ holds. By $\pi(i)$ we denote the i-th state on the path. By π^i we denote π's i-th suffix, i.e. $\pi^i = (s_i, s_{i+1}, \ldots)$.

2.7.3 Semantics

Informally, the meaning of temporal operators is

- $\mathbf{X}\varphi$: φ is true at next step,
- $\varphi_1\mathbf{U}\varphi_2$: φ_2 is true at some point, φ_1 is true until that time,
- $\mathbf{F}\varphi$: φ will become true at some point in the future, and
- $\mathbf{G}\varphi$: φ is always true.

The satisfaction $\models$ of a formula φ in LTL is with respect a pair of a Kripke structure and a path K, π and is defined inductively as the following

- $K, \pi \models \top$ is always satisfied

- $K, \pi \vDash \bot$ is is never satisfied
- $K, \pi \vDash p$ if and only if $p \in L(\pi_0)$, i.e. an atomic proposition is satisfied when it is labelled at the first element $\pi(0)$ of the path π
- $K, \pi \vDash \neg\varphi$ if and only if $K, \pi \nvDash \varphi$.
- $K, \pi \vDash \varphi \vee \psi$ if and only if $K, \pi \vDash \varphi \vee K, \pi \vDash \psi$
- $K, \pi \vDash \mathbf{X}\varphi$ if and only if $K, \pi(1) \vDash \mathbf{X}\varphi$
- $K, \pi \vDash \varphi\mathbf{U}\psi$ if and only if $\exists i\,.\,(K, \pi(i) \vDash \psi) \wedge (\forall j < i\,.\,(K, \pi(j) \vDash \varphi))$
- $K, \pi \vDash \mathbf{F}\varphi$ if and only if $\exists i\,.\,K, \pi(i) \vDash \varphi$
- $K, \pi \vDash \mathbf{G}\varphi$ if and only if $\forall i\,.\,K, \pi(i) \vDash \varphi$

Because of its unique features, temporal logics is a effective tool to formalize two main classes of properties of systems [Lam83, MP90]:

- A safety property stipulates that "bad things" do not happen during execution of a program or this property mest be satisfied by all reachable states of a given system.
- A liveness property stipulates that "good things" eventually do happen.

States which violate a given property are usually called bad states.

Example 2.5, 2.7 and 2.6 describe how to describe system behavior by LTL. Notice that two first examples are for safety properties and the last one is a liveness property.

Example 2.5. Process A and B are never both in their critical sections at the same time (mutual exclusion).

$$\mathbf{G}\,(\neg inCS_A \vee \neg inCS_B)$$

Example 2.6. Once red, the light cannot become green immediately.

$$\mathbf{G}\,(red \Rightarrow \neg\mathbf{X}green)$$

Example 2.7. Traffic light is green infinitely often.

$$\mathbf{GF}green$$

2.8 Conclusions

Specifying and reasoning about a distributed algorithm is a difficult work. In order to give the reader an idea of how to do that, we have given a compact introduction of LTL. The reason we choose LTL is that the underlying logic of TLA$^+$, Temporal Logic of Action, is a variant of LTL.

An SMT solver is a tool for deciding the satisfiability of an SMT problem which is formalised with equality and many theories in MS-FOL. At the moment, researchers usually use an SMT solver when applying predicate abstraction. In order to help the user understand the capability of an SMT solver, we have mentioned fundamental concepts of first-order logic (and common background theories) and described internal mechanism of advanced SMT solvers. Moreover, we have explained why we choose Z3 as a back-end prover in our work.

However, we cannot directly evaluate a TLA$^+$ expression with an SMT solver. One reason is that TLA$^+$ contains many complex constructs which do not have counterparts in SMT-LIB. A traditional way to remove complex structures is to repeatedly apply rewriting rules which replace subterms of a given expression with equal terms until the (possible) simplest form is obtained. In this chapter, we have recalled crucial properties which a TRS should have and have explained how to create a “good” TRS from a set of equations.

CHAPTER 3

TLA$^+$ Language and its Toolbox

TLA$^+$ is a formal specification language introduced by Leslie Lamport originally for specifying and reasoning about concurrent and reactive systems [Lam02]. This language is based on the linear-time temporal logic of actions (TLA) for the characterization of the dynamic system behavior and a variant of standard Zermelo-Fraenkel set theory with the axiom of choice (ZFC) for the description of data structures. It has been used successfully for many industrial projects at Microsoft, Compag, Intel and Amazon [BL02, TY02, New14]. In order to know how to make best use of TLA$^+$ and its supporting tools, we recommend the user to read Lamport's book "Specifying Systems" [Lam02]. This section outlines only its underlying logic, the specification structure and its toolbox.

Chapter Overview Section 3.1 explains the logical aspect of TLA$^+$ over which we will work in the rest of this document. In Section 3.2, we use a real example to describe a TLA$^+$ specification to introduce some basic concepts. In Section 3.3, we discuss its supporting tools, including SANY, TLAPS and TLC.

3.1 Underlying Logic

The logical foundations of TLA$^+$ are TLA and an untyped variant of ZFC set theory. An expression in TLA$^+$ may be in one of four basic levels *constant*, *state*, *transition* and *temporal*, which are described in detail later. We start by defining the syntax and semantics of Temporal Logic of Actions. Then, we describe the set-theoretical aspect of TLA$^+$ with the choice operator and gradually the first-order fragment over which the set theoretical axioms are defined. Finally, we presents extensions in TLA$^+$ which are functions, records, tuples, IF/THEN/ELSEs and CASE.

All definitions in the following are from [Lam94, Lam02].

3.1.1 TLA

We assume given two non-empty and disjoint collections

- Val of *value* symbols, such as numbers, strings and sets but not Boolean values (true, false), and
- Var of *variable* symbols.

There are two kinds of variables: flexible and rigid ones. The former are is declared by a `VARIABLES` statement, and the latter are variables whose value is the same in every state of the system's behavior which is defined formally in the following paragraph. Var is considered as fixed collections of built-in and user-defined symbols in a specification.

In the following, we give a compact introduction of operators in TLA$^+$. Operators in TLA$^+$ are distinguished into two different categories: constant operators and nonconstant operators. Constant operators are usually popular operators, nonconstant operators are what distinguish TLA$^+$ from ordinary mathematics.

Constant operators are ones of ordinary mathematics, having nothing to do with TLA or temporal logic. For instance, logical connectives or set operators are constant operators. Constructs for functions, records and tuples are also constant operators. A main feature of constant operators is that we can understand their meanings without considering their arguments. For the complete list of constant operators, please read pages 268 and 269 in Lamport's book [Lam02].

There are two categories of nonconstant operators: action operators in Table 3.1 and temporal operators in Table 3.2 [1]. To understand these operators, we need to think about their arguments. Here we bring only the definition of *basic* expressions which contains built-in TLA$^+$ operators, declared constants and declared variables [2].

Table 3.1: Action operators

e'	[The value of e in the final state of a step]
$[A]_e$	$[A \lor (e' = e)]$
$\langle A \rangle_e$	$[A \land (e' \neq e)]$
ENABLED A	[An A step is possible]
UNCHANGED e	$[e' = e]$
$A \cdot B$	[Composition of actions]

A constant expression is one containing only constant operators and declared constants. A *computation* (or behavior) of a system in a TLA specification is formalized as a sequence

[1]In Table 3.1 and 3.2, e is a state function, A is an action and F, G are temporal formula. These terms are explained in following paragraphs.

[2]TLA$^+$ allows the user define new operators in modules and to construct nonbasic expressions. Please read Lamport's book for more information.

Table 3.2: Temporal operators

$[]F$	[F is always true]
$<> F$	[F is eventually true]
$\mathrm{WF}_e(A)$	[Weak fairness for action A]
$\mathrm{SF}_e(A)$	[Strong fairness for action A]
$F \ \ > G$	[F leads to G]

of states, where a *state* of the system is an assignment of values to variables and is usually represented as a Boolean formula or a predicate . A pair of consecutive states, s_i and s_{i+1} say, is named a *step*, denoted $s_i \rightarrow s_{i+1}$. The prime $(\prime)$ operator is used to differentiate the values of variables in a step. Considering a given step $S : s_i \rightarrow s_{i+1}$ and assuming a variable v on S, the unprimed occurrence (v) refers to its value in s_i while the primed one (v') refers to its value in s_{i+1}. We write an *initial predicate* that specifies the possible initial values of variables, and a *next-state relation* that specifies how the value of variables can change in any step.

A state function is an ordinary expression (one without priming, action or temporal operators listed in Table 3.1, 3.2) that can contain unprimed variables, declared constants, constant operators and ENABLED expressions. In other words, a state function is a mapping from states to values. A *state predicate* is a Boolean-valued state function.

A *transition function* is an expression built from state functions using the priming operator $(\prime)$ and the other action operators of TLA$^+$ listed in Table 3.1. A transition function assigns a value to every step, where a step is a pair of states. For example, if step S is such that $v = \{0\}$ in s_i and $v = \{1\}$ in s_{i+1} , the transition function $[v' \cup v]$ equals $\{0, 1\}$ on S. Finally, an *action* is defined as a Boolean-valued transition function, such as $v \subseteq v'$ [3].

A *temporal* formula is Boolean assertion about a behavior, where behavior is a sequence of states. Syntactically, a temporal formula is defined inductively to a state predicate or a formula having one of the forms shown in 3.2. A behavior satisfies a formula F if F is a true assertion of this behavior.

An *invariant Inv* of a specification *Spec* is a state predicate such that $Spec \Rightarrow \Box Inv$ is a theorem. A variable v has type T in a specication *Spec* if and only if $v \in T$ is an invariant of *Spec*.

In TLA, an expression has one of four basic levels, which are numbered 0, 1, 2 and 3. Assume two variables x, y are declared by VARIABLES, a constant c is declared by CONSTANT and two symbols $-, <$ have their usual meanings in arithmetic. These levels are described below

0. A *constant*-level expression is a constant; it has only constants and constant operators. Example: $c - 1$.

[3] We do not permit quantification over flexible variables in state functions and actions.

1. A *state*-level expression is a state function; it may have constants, constant operators and unprimed variables. Example: $x - c$.
2. A *transition*-level expression is a transition function; it may have anything except temporal operators. Example: $x - y' < c$.
3. A *temporal*-level expression is a temp oral formula; it may have any TLA operator. Example: $\Box [x' > y - c]_{\langle x,y \rangle}$.

3.1.2 TLA+ set theory

In this subsection, we describe the set theoretical aspect of TLA+ with common constructs and operators. For other constructs, please read Lamport's book [Lam02] [4].

TLA+ is based on ZFC set theory, in which every value in TLA+ is a set, even a natural number such as 0 [5]. In set theory, the operator $\in$ is considered as a primitive, undefined operators. We could define all the other set operators by the opeartor $\in$, predicate logic and the operator CHOOSE .

The operator CHOOSE in TLA+ is Hilbert's choice operator ϵ, written $\mathsf{CHOOSE}\ x : P(x)$. This expression denotes an arbitrary but fixed value x such that $P(x)$ is true. If no such x exists, then the expression has a completely arbitrary value.

The semantics of CHOOSE are expressed by the following two rules:

1. $\exists x : P(x) \equiv P(\mathsf{CHOOSE}\ x : P(x))$
2. $\forall x : P(x) = Q(x) \Rightarrow (\mathsf{CHOOSE}\ x : P(x)) = (\mathsf{CHOOSE}\ x : Q(x))$

for any operators P and Q. We know nothing about the value chosen by CHOOSE except what we can deduce from these rules. The second one expresses the equality of CHOOSE , that is, it assigns the same witness value to equivalent formulas P and Q or CHOOSE is not a non-deterministic operator. Moreover, the expression $\mathsf{CHOOSE}\ x : \mathsf{FALSE}$ and all its equivalent forms represent a unique value.

With the operator $\in$, predicate logic and the operator CHOOSE , we can define set union as following:

$$S \cup T \triangleq \mathsf{CHOOSE}\ U : \forall x : (x \in U) \equiv (x \in S) \lor (x \in T)$$

In standard set theory, sets are constructed from axioms that state their existence. Here, we add the set constructs as primitive objects of the language. The set objects are the empty set $\{\}$, the power set SUBSET, the generalized union UNION, two forms of set comprehension $\{x \in S : p\}$ and $\{e : x \in S\}$ [6]. Primitive operators are which we can define

[4] Complex constructs can be inductively defined on those which we introduce in this subsection.

[5] A number 0 is a set but we do not know exactly what elements of 0 are.

[6] The general construct is $\{e : y_1 \in S_1, \ldots, y_n \in S_n\}$ and the user can find its definition in Lamport's book.

mathematically in terms of the rules that they satisfy. Other set operators such as $\cup$ or $\cap$ are also considered to be primitive.

Here is the list of defining rules. A rule with "is defined by" shows that the operator in the left-hand side is taken as a primitive one [7].

$$\begin{aligned}
S = T &\triangleq \forall x : (x \in S) \equiv (x \in T)\\
e_1 \neq e_2 &\triangleq \neg(e_1 = e_2)\\
e \notin S &\triangleq \neg(e \in S)\\
S \subseteq T &\triangleq \forall x : (x \in S) \Rightarrow (x \in T)\\
S \cup T &\text{ is defined by } \forall x : (x \in (S \cup T)) \equiv (x \in S) \vee (x \in T)\\
S \cap T &\text{ is defined by } \forall x : (x \in (S \cup T)) \equiv (x \in S) \wedge (x \in T)\\
S \setminus T &\text{ is defined by } \forall x : (x \in (S \cup T)) \equiv (x \in S) \wedge (x \notin T)\\
\{\} &\text{ is defined by } \forall x : x \notin \{\}\\
\{e\} &\text{ is defined by } \forall x : (x \in \{e\}) \equiv (x = e)\\
\{e_1, \dots, e_n\} &\triangleq \{e_1\} \cup \dots \cup \{e_n\}\\
\{x \in S : p\} &\text{ is defined by } \forall y : (y \in \{x \in S : p\}) \equiv (y \in S) \wedge p[x \leftarrow y]\\
\{e : x \in S\} &\text{ is defined by } \forall y : (y \in \{e : x \in S\}) \equiv (\exists x \in S \wedge e = y)
\end{aligned}$$

The defining rule of $\{x \in S : p\}$ requires that x is a bound identifier, S is outside the scope of the bound identifier and y is a fresh variable [8].

In addition to the unbounded form of quantifiers and CHOOSE , TLA$^+$ supports the bounded form which is defined by

$$\begin{aligned}
\forall x \in S : P(x) &\triangleq \forall x : x \in S \Rightarrow P(x)\\
\exists x \in S : P(x) &\triangleq \exists x : x \in S \wedge P(x)\\
\mathsf{CHOOSE}\ x \in S : P(x) &\triangleq \mathsf{CHOOSE}\ x : x \in S \wedge P(x)
\end{aligned}$$

3.1.3 Other constructs

Conditional expressions

TLA$^+$ offers two conditional constructs IF THEN ELSE and CASE with their obvious meanings like in programming languages.

Numbers

TLA$^+$ allows the user to work with arithmetic expressions. *Nat*, *Int* and *Real* denotes the set of natural number, of integer numbers and of real numbers, respectively. In addition

[7] This operator appears in both the left-hand side and the right-hand side, i.e. $S \cup T$

[8] x can be a tuple of bound identifiers.

to common operators, it offers the operator .. which is the interval between two integer numbers, that is, $a \,..\, b \triangleq \{n \in Int : a \leq n \wedge n \leq b\}$ where $a \leq b$.

Strings

TLA+ defines a string to be a tuple of characters. However, the semantics of TLA+ does not specify what a character is and exactly what characters may appear in a string is system-dependency [Lam02].

Functions

Functions are in the extension of TLA+ set-theoretic fragment. In principle, all well-formed expressions denote sets, but some of them are used as functions, as they are called in TLA+ . To explain functions, we introduce meanings of 4 constructs:

$$f[e] \quad \mathsf{DOMAIN}\, f \quad [S \rightarrow T] \quad [x \in S \mapsto e]$$

where x is an identifier.

A function f in TLA+ has a *domain*, written $\mathsf{DOMAIN}\ f$, and it maps to each element x of its domain one unique value, written $f[x]$. For any $x \in S$, an expression $f[x]$ is legal but its value is unspecified. The *range* of a function f is the set of all values of the form $f[x]$ with x in $\mathsf{DOMAIN}\ f$.

TLA+ offers a convenient way to describe explicitly a function f with the construct

$$f \triangleq [x \in S \mapsto e]$$

The above expression means that the the domain of function f with domain S and f equals the value obtained by substituting v for x in e, for any $v \in S$. For example, $f_1 = [x \in \{1,2\} \mapsto 0]$ depicts the function f_1 with domain $\{1,2\}$ such that $f_1[1] = 0$ and $f_1[2] = 0$.

For any sets S and T, the set of all functions f whose domain equals S and whose range is any subset of T is written $[S \rightarrow T]$, that is, $\mathsf{DOMAIN}\ f = S$ and $f[v] \in T$ for all $v \in S$. Note that it is possible to quantify over (terms representing) functions. For instance, an expression

$$\forall f \in [\{1,2\} \rightarrow \{1,2\}].\mathsf{DOMAIN}\, f = \{1,2\}$$

is well-formed.

In addition, TLA+ has the construct for function-update $[f\ \mathsf{EXCEPT}\ ![d] = e]$ depicts the function $\hat{f}$ equal to f except that $\hat{f}[d] = e$.

In TLA+, to check whether a set g is a function or not, we can use the following operator [Lam02]

$$IsAFcn(g) \triangleq g = [x \in \mathsf{DOMAIN}\ g \mapsto g[x]],$$

We have that $IsAFcn(g)$ is true if and only if g is a function.

Now, we can define the axiom of function extensionality as follows:

$$\begin{aligned} f = [x \in S \mapsto e] \Leftrightarrow\ & \land IsAFcn(f) \\ & \land \mathsf{DOMAIN}\ f = S \\ & \land \forall y \in S : f[y] = e[x \leftarrow y] \end{aligned} \qquad (3.1)$$

Note that these above constructs are primitive and there is no separate defining rule for the DOMAIN operator. For further information about functions and its constructs, such as recursive functions, we refer the reader to Lamport's book [Lam02].

Tuples and records

Unlike programming languages, tuples and tuples in TLA$^+$ are functions.

An n-tuple $\langle e_1, \ldots, e_n \rangle$ is also a function whose domain is a set of n integers $\{1, , \ldots, n\}$ and which maps 1 to e_1, 2 to e_2 and so on.It means $\langle e_1, \ldots, e_n \rangle[i] = e_i$ where $1 \leq i \leq n$. TLA$^+$ provides the Cartesian product operator $\times$ of ordinary mathematics to construct a set of tuples. For example, $A \times B \times C$ is the set of all 3-tuples $\langle a, b, c \rangle$ such that $a \in A, b \in B$ and $c \in C$. Note that the operator $\times$ is not associative and the 0-tuple $\langle\rangle \triangleq [x \in \{\} \mapsto \{\}]$ is the unique function having an empty domain.

A record is a function whose domain is a finite set of strings. For example, a record msg with p, val, rnd fields is a function whose domain is the set of the three strings "p", "val" and "rnd". The expression $msg.val$ is an abbreviation for msg["val"]. To construct a record msg, the user can write

$$[p \mapsto 2, val \mapsto 1, rnd \mapsto 1]$$

which can be written

$$\begin{aligned} [i \in \{\text{“p”}, \text{“val”}, \text{“rnd”}\} \mapsto\ & \mathsf{IF}\ i = \text{“p”}\ \mathsf{THEN}\ 2\ \mathsf{ELSE} \\ & \qquad\qquad \mathsf{IF}\ i = \text{“val”}\ \mathsf{THEN}\ 1\ \mathsf{ELSE}\ 0] \end{aligned}$$

In general, record construction is defined by the following rule

$$[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$$

and the below rule is to define the set of records.

$$[h_1 \in S_1, \dots, h_n \in S_n]$$

The EXCEPT construct is also for records. Note that the TLA$^+$ definition of records as functions makes it possible to manipulate them in ways that have no counterparts in programming languages [Lam02].

3.1.4 TLA$^+$ semantics

Because of its untyped properties, TLA$^+$ accepts expressions like $2 \wedge$"abc" or $\mathsf{FALSE} \Rightarrow \langle 5 \rangle$. We must therefore decide how to interpret them. The standard semantics of TLA+ offers three different ways to decide the meaning of these expressions which are called the conservative, moderate and liberal interpretations.

In the *conservative interpretation*, we do not completely specify the value of an expression like $2 \wedge$ "abc". It could equal $\sqrt{3}$ or any other value. Moreover, it need not to be equal to "abc" $\wedge\, 2$. Hence, the ordinary laws of logic, such as the of two Boolean operators $\vee$ and $\wedge$, are valid only for Boolean values.

In the *liberal interpretation*, the value of $2 \wedge$ "abc" is specified to be a Boolean value, however this interpretation does not mention whether the value of this expression equals TRUE or FALSE. However, all the ordinary laws of logic, such as the commutation or association, are valid. Hence, $2 \wedge$ "abc" equals "abc" $\wedge\, 2$. More precisely, any tautology of propositional or predicate logic is valid, even if the value of some its sub-formula is not a Boolean one. As a result, the liberal approach is sound.

The only difference between conservative and liberal interpretations the use of Boolean-valued functions. For instance, suppose we define the function *isReal* by

$$\mathit{isReal} \triangleq [r \in \mathit{Real} \mapsto \mathsf{TRUE}]$$

so $\mathit{isReal}[r]$ equals TRUE for all r in *Real*. Now, consider the formula

$$\forall x : (x \in \mathit{Real}) \Rightarrow \mathit{isReal}[x] \tag{3.2}$$

which asserts that $(x \in \mathit{Real}) \Rightarrow \mathit{isReal}[x]$ is true for all x , including, for example, $x =$ "a". Let x be "a", we obtain a formula ("a" $\in \mathit{Real}) \Rightarrow \mathit{isReal}[$"a"$]$. While this formula is TRUE in the liberal interpretation, its value is unspecified in the conservative interpretation. Therefore, the formula 3.2 is TRUE in the liberal interpretation, but its value is unspecified in the conservative interpretation. If we are using the conservative interpretation, instead of 3.2, we should write

$$\forall x : (x \in \mathit{Real}) \Rightarrow \mathit{isReal}[x] = \mathsf{TRUE} \tag{3.3}$$

The formula 3.3 equals TRUE in both interpretations.

The *moderate interpretation* lies between the conservative and liberal interpretations. Not-so-silly formula which involves TRUE and FALSE like $\textsf{FALSE} \Rightarrow$ “a” have their Boolean expected values. For example, $\textsf{FALSE} \Rightarrow$ “a” equals TRUE and $\textsf{FALSE} \land$ “a” equals FALSE. However, the value of $2 \land$ “abc” is still completely unspecified. This causes two problems in the moderate interpretation:

1. The laws of logic do not hold unconditionally in this approach. The formulas $p \lor q$ and $q \lor p$ are equivalent only if p and q are both Boolean expressions, or if one of them equals TRUE.
2. When using the moderate interpretation, we still have to check that all the values of subformulas are Booleans before applying any of the ordinary rules of logic in a proof.

In our work, we prefer the conservative interpretation. For expressions with unspecified values, we will give the user warnings.

3.2 Specification

In TLA$^+$, specifications are formulas and are organised in modules. A module usually consists of three sections which are declarations (of parameters and variables), definitions (of operators) and assertions (of assumptions and theorems). However, this structure is conventional, but not mandatory. TLA$^+$ only requires that an identifier must be declared or defined before it is used, and that it cannot be reused, even as a bound variable, in its scope of validity. Horizontal lines are usually added between different sections of a module to make it easier to read, but they have no semantic content. In general, a module looks like

```
┌──────── MODULE bcastFolklore ────────┐
 declaration
├──────────────────┤├─────────────────┤
 definition
├──────────────────┤├─────────────────┤
 assertion
└──────────────────────────────────────┘
```

Declarations These describe constant parameters and variables in the system by commands CONSTANTS and VARIABLES. There are typically a dozen or so declared identifiers. For example,

Definitions These define mathematical operators used to describe operations specific to the particular system. They take the form $Op\,(arg_1, \ldots, arg_n) \triangleq e$. For example, a specification of a distributed system can have user-defined operators *Receive* and *Broadcast*. A specification can inherit definitions from others by a command EXTENDS. Four interesting and important definitions are

1. The *initial predicate* describes the possible initial values of variables, or possible initial states of the system. It is a conjunction of formulas $x = \ldots$ or $x \in \ldots$ for each variable x, where the $\ldots$ is usually a simple constant expression. It is usually named *Init*.

2. The *next-state action* describes the system's transitions over primed and unprimed variables. Usually It is defined as a disjunction of sub-actions, each system step. For example, in the specification *bcastFolklore*, assume that $N = 2$, so there are two sub-actions $Step(1)$ and $Step(2)$. It is usually named *Next*.

3. *Liveness* defines a temporal formula specifying the liveness properties of the system, usually in terms of fairness conditions on sub-actions of the next-state action.

4. The *specification*, usually named *Spec*, is the one-line definition

$$Spec \triangleq Init \land \Box[Next]_{vars} \land Liveness$$

   where $\Box$ is the ordinary "always" operator of linear-time temporal logic, *vars* is a tuple representing all variables in the specification and $[Next]_{vars}\,s$ abbreviates $Next \lor$ UNCHANGED *vars*. It means the specification is always represented by a single temporal formula.

Assertions are assumptions and theorems which are introduced by ASSUME, THEOREM, ... For instance, in the specification *bcastFolklore*, the number of crashed processes is less than a half of the number of processes. In our work, we do not focus on assertions.

The following TLA$^+$ specification is the encoding of Chandra's algorithm for reliable broadcast by message diffusion (BcastFolklore) [CT96]. This encoding was provided by my supervisors and used in our experiment. For the description of this algorithm, we refer the user to Chapter 7.

```
──────────────────── MODULE bcastFolklore ────────────────────

EXTENDS Naturals, FiniteSets

CONSTANTS N, T, F

VARIABLE pc, rcvd, sent, nfailed
```

$\textsc{assume}\ N \in Nat \land T \in Nat \land F \in Nat$

$\textsc{assume}\ (N > 2 * T) \land (T \geq F) \land (F \geq 0)$

$P \triangleq 1 \,..\, N$ all processess, including the faulty ones

$Corr \triangleq 1 \,..\, N$ correct processes

$M \triangleq \{\text{“ECHO”}\}$

$vars \triangleq \langle pc, rcvd, sent, nfailed \rangle$

$Receive(self) \triangleq$

$\quad \exists\, r \in SUBSETPM :$

$\qquad \land\ r \subseteq sent$

$\qquad \land\ rcvd[self] \subseteq r$

$\qquad \land\ rcvd' = [rcvd\ \textsc{except}\ ![self] = r]$ receive set ”r“ of msgs

$UponV1(self) \triangleq$

$\quad \land\ pc[self] = \text{“V1”}$ if a process ”has received a msg from a bcasting process and has not sent ($ECHO$)“

$\quad \land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“AC”}]$ it accepts and sends ($ECHO$) to all

$\quad \land\ sent' = sent \cup \{\langle self, \text{“ECHO”} \rangle\}$

$\quad \land\ nfailed' = nfailed$ a number of crashed processes does not change

$UponCrash(self) \triangleq$

$\quad \land\ nfailed < F$ if a number of crashed processes $< F$ and

$\quad \land\ pc[self] \neq \text{“CR”}$ this process is correct, it will be crashed

$\quad \land\ nfailed' = nfailed + 1$ increase a number of crashed processes

$\quad \land\ pc' = [pc\ \textsc{except}\ ![self] = \text{“CR”}]$ update labels of processes

$\quad \land\ sent' = sent$ message channel does not change

$UponAccept(self) \triangleq$

$\quad \land\ (pc[self] = \text{“V0”} \lor pc[self] = \text{“V1”})$ if a process ”has not receive any msg“ or

$\land\ rcvd'[self] \neq \{\}$ ”has received a msg from a bcasting process and has not sent (*ECHO*)“

$\land\ pc' = [pc \text{ EXCEPT } ![self] = \text{“AC”}]$ it accepts and sends (*ECHO*) to all

$\land\ sent' = sent \cup \{\langle self, \text{“ECHO”}\rangle\}$

$\land\ nfailed' = nfailed$ a number of crashed processes does not change

an action

$$\begin{aligned} Step(self) \triangleq\ & \land\ Receive(self) \\ & \land\ \lor\ UponV1(self) \\ & \quad \lor\ UponCrash(self) \\ & \quad \lor\ UponAccept(self) \\ & \quad \lor\ pc' = pc \land sent' = sent \land nfailed' = nfailed \end{aligned}$$

$Init \triangleq$

$\land\ sent = \{\}$ message channel is empty

$\land\ pc \in [Corr \to \{\text{“V0”}, \text{“V1”}\}]$ process can be labeled as ”has not received any msgs“ or ”has received a msg from a bcasting process and has not sent (*ECHO*)“

$\land\ rcvd = [i \in Corr \mapsto \{\}]$ every process has not received any msg

$\land\ nfailed = 0$ no process crashes

the Next predicate

$\land\ rcvd = [i \in Corr \mapsto \{\}]$

$$Next \triangleq (\exists\, self \in Corr : Step(self))$$

specification

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

type invariant

$$\begin{aligned} TypeOK \triangleq\ & \land\ sent \subseteq P \times M \\ & \land\ pc \in [Corr \to \{\text{“V0”}, \text{“V1”}, \text{“AC”}, \text{“CR”}\}] \\ & \land\ rcvd \in [Corr \to \text{SUBSET } (P \times M)] \end{aligned}$$

system properties

$Unforg \triangleq (\forall\, self \in Corr : (pc[self] \neq \text{“AC”}))$

$UnforgLtl \triangleq (\forall\, i \in Corr : pc[i] = \text{“V0”}) \implies \Box(\forall\, i \in Corr : pc[i] = \text{“AC”})$

$CorrLtl \triangleq (\forall\, i \in Corr : pc[i] = \text{“V1”}) \implies \Diamond(\exists\, i \in Corr : pc[i] = \text{“AC”})$

$RelayLtl \triangleq \Box((\exists\, i \in Corr : pc[i] = \text{“AC”}) \implies \Diamond(\forall\, i \in Corr : pc[i] = \text{“AC”}))$

3.3 TLA$^+$ Toolbox

The TLA$^+$ Toolbox is an integrated development environment (IDE) for writing specifications and running tools to check them [Lam02]. The main tools include the parser and syntax checker SANY, the interactive theorem prover TLAPS [CDL$^+$12] and the model checker TLC c.

3.3.1 SANY

SANY is a parser and syntax checker for TLA$^+$ specifications. Therefore, SANY is responsible for creating the abstract-syntax tree and semantic tree of the specification.

3.3.2 TLAPS

The TLA$^+$ specification language was extended to permit writing hierarchically structured proofs [CDLM10] and the tool TLA+ Proof System (TLAPS) has been developed to deductively verify specifications, their properties and proofs formally [CDL$^+$12]. It is an interactive proof environment which is built around an application called the Proof Manager. TLAPS is built around an application called the Proof Manager. The manager first interprets a TLA$^+$ proof as a collection of proof obligations. Then, the manager sends proof obligations to back-end theorem provers to prove. Finally, if possible, a verifier checks the proof generated by the back-end prover to provide complete machine-checking of TLA+ proofs. At the time of starting this work, TLAPS was based on three available back-ends with different capabilities Zenon, Isa and SMT.

1. Zenon [BDD07] is a tableau prover for FOL.
2. Isa is the automatic tactic auto of the Isabelle prover [NPW02].
3. SMT is the baseline SMT solver. The default one is CVC3, but it allows the user to use Z3 or Yices (only version 1) [BT07, dMB08, DDM06].

While TLAPS is a new tool, it is successfully used in some projects which includes the verification of the Paxos consensus algorithm [Lam11], the Memoir [PLD+11] security architecture and the Pastry [LMW12] algorithm. The safety-proof of a specification has the following structure [Lam02]:

$\textsc{lemma}\ \mathit{Spec} \implies \Box \mathit{SafetyProperty}$

(* Dijkstra's invariant implies correctness *)

$\langle 1 \rangle 1\ \mathit{Inv} \implies \Box \mathit{SafetyProperty}$

(* Dijkstra's invariant is (trivially) established by the initial condition *)

$\langle 1 \rangle 2\ \mathit{Init} \implies \mathit{Inv}$

(* Dijkstra's invariant is inductive relative to the type invariant *)

$\langle 1 \rangle 3\ \mathit{TypeOK} \land \mathit{Inv} \land [\mathit{Next}]_{\mathit{vars}} \implies \mathit{Inv}$

$\langle 1 \rangle q\ \mathit{QED}$

$\textsc{by}\ \langle 1 \rangle 1,\ \langle 1 \rangle 2,\ \langle 1 \rangle 3,\ \langle 1 \rangle q,\ \mathit{TypeOK}_{\mathit{inv}},\ \mathit{PTL}\ \textsc{def}\ \mathit{Spec}$

3.3.3 TLC

TLC is an explicit-state model checker for TLA+ specifications in the standard form

$$\mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{Liveness}$$

where *Liveness* is an option [YML99]. If *Liveness* is omitted, TLC checks invariants and "silliness" errors whose meaning is not defined by the semantics of TLA+ and deadlock which can be disabled. In general, TLC checks the specification by exploring all states satisfying 3.3.3 and looking for one in which a desired property is not satisfied or deadlock occurs. If the property is violated, TLC will show a minimal length trace that leads from an initial state to the bad state. TLC stops when it has examined all reachable states. Certainly, TLC may never terminate if this set of these states is infinite.

To work with TLC, the user must first create a model of the specification which contains values of all specification's constant parameters and states clearly what properties needing to check. Moreover, the user should add constraints to make the model become a finite-state system. For example, the specification in our example module *bcastFolklore* is not finite-state because:

- the set of processors can be arbitrarily large — even infinite, and

- the number of states could depend on the unspecified parameters N, T, and F.

Fortunately, it is not difficult to bound the number of states. The user needs to only assign particular values to the constants N, T, and F. Because of the constraints, TLC will check only states that appears in a behavior satisfying

$$Init \wedge \Box[Next]_{vars} \wedge \Box Constraint \wedge Liveness$$

and that are usually called the reachable ones.

A main feature of TLC is that it uses an explicit state representation because [YML99]:

- A symbolic method would need additional restrictions on the class of TLA$^+$ specifications TLC could handle.
- Explicit state representations seem to work at least as well for the asynchronous systems.
- It is burdensome to keep a symbolic representation on disk.

At the moment, TLC can check both safety and liveness properties. However, in the distributed mode, TLC cannot verify liveness properties since checking a liveness property is very expensive [Lam02]. Fortunately, liveness properties are rarely written in TLA$^+$ specification [BL02]. Moreover, we focus on only safety properties in this thesis. Therefore, here we describe how TLC checks safety properties.

Table 3.1 shows how TLC works. TLC first generates and checks all possible states satisfying the initial predicate, and then applies the breadth-first search algorithm for traversing the state space. If TLC finds a bad state, it will print an error trace and stop. To overcome the space limitation, TLC therefore keeps all data on disk, using main memory as a cache. To make the computation of next states simple and efficient, TLC rewrites the next-state relation as a disjunction of as many simple sub-actions as possible.

In experiments at Amazon, Newcombe et al. had 2 TB of local SSD storage to keep visited states [New14]. So, techniques TLC applies are not impractical as one can think.

Thanks to an explicit state representation, TLC can handle a critical fragment of TLA$^+$ that most people actually write [Lam02]. All values in this fragment are are built from the following four types of primitive values:

A TLC value is defined inductively to be either

1. a primitive value, or
2. a finite set of comparable TLC values [9], or

[9]Informally, two values in TLA$^+$ are comparable if the semantics of TLA$^+$ decides whether they are equal or not. For precise rules, please read the book [Lam02]

Algorithm 3.1: TLC's strategy to verify a safety property

```
input  : A TLA$^+$ specification $Spec$ and safety property $inv$
output : true or an error trace if a bad state exists

1  create an empty queue Q ;
2  while exists state $s$ s.t. $s$ satisfies $Init$ and $s$ is unvisited do
3      if $s$ violates $inv$ then
4          print a bad state $s$ ;
5      end
6      add $s$ to $Q$ ;
7      mark $s$ visited ;
8  end
9  while $queue \neq \{\}$ do
10     $s$ = dequeue $Q$ ;
11     if exists unvisited state $t$ is s.t. $(s, t)$ satisfies an action in then
12         mark $s$ as a father of $t$ ;
13         if $t$ violates $inv$ then
14             print an error trace ;
15         end
16         add $t$ to $Q$ ;
17         mark $t$ visited ;
18     end
19 end
20 return true ;
```

Booleans	The values TRUE and FALSE.
Integers	Values like 5 and -2.
Strings	Values like "a3b" .
Model Values	These are values, called model values, assigned to the *CONSTANT* parameters in the specification. Model values with different names are considered to be distinct.

3. a function f whose domain is a TLC value such that for all elements $x \in$ DOMAIN f, we have that $f[x]$ is also a TLC value.

TLC cannot evaluate

- quantifiers or CHOOSE expressions with infinite domains,
- any expression whose value is not a TLC value,
- a set-valued expression for an infinite set,
- a function-valued expression whose domain is infinite, and

- a recursive definition which causes an infinite loop.

At the moment, TLC has many optimization which allow TLC to evaluate an expression even when it cannot assess all sub expressions. For example, TLC can evaluate

$$[n \in Nat \mapsto n + 1][3]$$

which equals the TLC value 4, even though it cannot evaluate the function $[n \in Nat \mapsto n + 1]$ since its domain is an infinite set Nat.

Until now, TLC has been successfully applied to the verification of software and hardware systems in many industrial projects. We can mention several examples: cache-coherency protocols for microprocessors at Compaq and Intel [TY02, BL02], and fault-tolerant distributed algorithms at Amazon [New14], among others. Moreover, TLC is also used to construct the formal proof by checking the invariant. Before attempting to prove correctness of a TLA$^+$ specification, the user should check (small) finite instances with TLC.

3.3.4 Discussions of TLAPS and TLA

While writing proofs is a great reliable method to reason about properties of the system, it is a hard and extensive work.

1. Finding an inductive invariant is a difficult and error-prone task. To avoid useless attempt, the user should try to check their ideas in small cases by TLC. Unfortunately, TLC is still a explicit tool and needs a lot of time to check the invariant [Lam02].
2. TLAPS does not offer any reasoning mechanism about TLA$^+$ features which are recursive operators, real numbers, many temporal operators, quantification over tuples and set constructors using tuples, the operator ENABLED, the action composition and operators in the TLC standard module [Res]. In these cases, the user can write OMITTED to inform that TLAPS should skip these lemmas or believe they are true. However, it is not easy to know when the user should use OMITTED.
3. The back-end provers require a lot of guidance to reason about CASE constructs, strings, tuples, records, the CHOOSE operator, and complicated operators in arithmetic such as the division, modulus, or exponentiation [Res].
4. A full formal proof may contain thousands of lines. For example, the formal specification and proof of Memoir contain 61 top-level definitions, 182 LET-IN definitions, 74 named theorems, and 5816 discrete proof steps [DLP$^+$11].

For the above reasons, it is difficult to use TLAPS in some cases. For example, Newcombe et al. tried to verify their critical algorithms by writing formal proofs but finally, he said "we doubt that we would use incremental formal proof as a design technique even for those algorithms" [New14]. In these cases, TLC is preferred.

The main advantage of model-checking is that it requires much less effort and less expertise in the verification domain from the user than writing a fully formal proof of correctness. Typically, the user needs to add constraints to make the state space, presses some buttons, and waits for results. The greatest weakness of this approach is the state-space explosion problem which makes it not scalable for industrial-strength specifications. To overcome it, the user usually needs to construct suitable abstractions of the system, which can reduce the number of states by mapping many concrete states to an abstract one. A resulting model is enough small to effectively be verified. However, at the moment, TLC does not support any abstract techniques and therefore, it cannot check many real projects.

3.4 Conclusions

In this section, first we have introduced the mathematical aspects of TLA$^+$

i. the temporal logic of actions (TLA) for the characterization of the dynamic system behavior, and

ii. a variant of standard Zermelo-Fraenkel set theory with the axiom of choice (ZFC) for the description of data structures.

We have also described the typical structure of a TLA$^+$ specification. A TLA$^+$ specification usually contains three parts: declarations of constants and variables, definitions of operators and assertions of system behavior.

We have discussed the TLA$^+$ Toolbox, an IDE for writing and verifying specifications. The main tools include the parser and syntax checker SANY, the interactive theorem prover TLAPS [CDL$^+$12] and the model checker TLC. Finally, we have made a compact comparison between two ways to prove the correctness of a TLA$^+$ specification: automated deduction with TLAPS and model checking with TLC.

CHAPTER 4

Model Checking

In this chapter, we describe a well-known technique for system verification, called model checking which applies a transition system and checks whether the behavior of the system model satisfy its specification. Model checking is a state-based method and it suffers from the infamous state-space explosion problem. To overcome the obstacle, researchers in model checking have applied many techniques such as symbolic representations, the partial order reduction, symmetry or abstraction. We focus on predicate abstraction which maps a system state to a vector of predicates, called an abstract state [GS97]. Notice that many concrete states can be mapped to the same bit vector. Unfortunately, constructing a full transition system is expensive. Therefore, instead of building and verifying a "complete" (abstract) model which represents explicitly all states and transition relations, we produce a formal proof of a safety property based on the IC3 algorithm.

Chapter overview Section 4.1 introduces basic ideas of model checking. Section 4.2 describes one of the most well-known techniques, called predicate abstraction, to reduce the state-space problem. Section 4.3 shows our preliminary experiments with NuSMV2, one of popular model checking tools. Section 4.4 presents how to construct an incremental proof based on the IC3 algorithm and predicate abstraction.

4.1 Overview

Errors are inevitable and cause serious problems in computer software programs, computer hardware designs and computer systems. The US National Institute of Standards and Technology (NIST) has estimated that programming failures pose a significant cost, around $60B annually, to the US economy [New02]. Moreover, software developers may spend half of their time fixing bugs and making code work. A great deal of research effort has been and is devoted to develop methods for error elimination. In the early 1980s, *temporal-logic model checking algorithms* were introduced to solve non-trivial verification problems automatically [CE81, QS82]. These algorithms were based on a combination of the state-exploration approach with temporal logic.

Applying model checking to a program consists of three main steps [CGP99] :

- **Modelling.** The first task is to transform a program into an input format accepted by a model checker. In many cases, the use of abstraction is required to remove irrelevant or unimportant information. For example, the model checker TLC accepts only specification written in the TLA$^+$ language.
- **Specification.** The second task is to assert properties that the program must satisfy. Nowadays, temporal logic is widely used to formalize the behavior of the system. In our work, TLA$^+$, a well-known variant of temporal logic, is used to formalize both system behavior and desired properties.
- **Verification.** Ideally the verification is completely automatic. However, in practice it often involves human assistance in constructing the abstraction, analyzing error traces and modifying the system.

Definition 4.1. Give a Kripke structure $M = (S, S_0, R, L)$ that represents a finite-state system and a temporal formula φ that formalize a desired property, the model checking problem is to find the set of all states $s \in S$ that satisfy φ, i.e. the set $\{s \in S \mid M, s \vDash \varphi\}$ [CGP99].

We say that a state s is a *bad state* if it violates a given property.

Compared to deductive verification based on theorem proves, model checking has a number of advantages

- No proofs. The user does not need to make a correctness proof which may require expert knowledge and months of the user's time working. In principle, all that is necessary for the user is to enter a description of the program to be verified, to give the specification to be checked, to press the "run" key and wait for results. The checking process is automatic.
- Diagnostic counterexamples. If the specification is violated, the model checker will produce a counterexample execution trace that shows why the desired property does not hold. The counterexamples bring real value in debugging complex systems.
- No problem with partial specifications. It is unnecessary to completely specify the system before checking properties with a model checker. Hence, model checking can be applied many times during the design of a complex system.

However, the *state-space explosion problem* is a major challenge in this approach. In real projects, the number of global system states of a concurrent system with many processes or complicated data structures can be enormous. Much of the research in this area is targeted at reducing the state-space of the model and many techniques have been introduced: such as symbolic representations, methods related to pushdown automata, symmetry or abstraction. Thank to the help of the methods, state-of-the-art model checkers now can solve problems in real complex systems, even infinite-state systems [Eme08, JM09].

In theory, model checking can verify both safety and liveness properties. However, the complexity of model checking liveness properties is inherently much more enormous than that of checking safety properties. Engineers therefore usually do not even write the liveness property. Moreover, only temporal operator $\Box$ usually appears in TLA$^+$ specifications [BL02]. Therefore, here we focus on only safety properties.

4.2 Predicate Abstraction

Abstraction is probably the most significant method to copy with the state-space explosion problem. Abstraction defines a relationship between the states of the concrete system and the states of a "reduced" system. To construct a smaller model, many concrete states are usually mapped to one abstract state. Abstraction is usually performed on a high level description of the system, before building the model for the system. Hence, the construction of the "prohibitively large" model is prevented.

The "reduced" model is usually constructed based on two techniques the cone of influence reduction and data abstraction [CGP99]. The cone of influence reduction eliminates variables that do not influence the variables in the specification. Data abstraction finds a mapping between the actual data values in the system and a small set of abstract data values. Data abstraction is based on the observation that the specifications of systems that include data paths usually involve fairly simple relationships among the data values in the system.

At the moment, there are many ways to construct an abstract model. *existential abstraction* is one of the most well-known technique [CGL94].

Definition 4.2. A model $\widehat{M} = \{\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L}\}$ is an existential abstraction of $M = (S, S_0, R, L)$ with respect to $\alpha : S \rightarrow \hat{S}$ if and only if

- $\exists s \in S_0 \, . \, \alpha(s) = \hat{s} \Rightarrow \hat{s} \in \widehat{S_0}$
- $\exists (s_0, s_1) \in R \, . \, \alpha(s) = \hat{s} \wedge \alpha(s_1) = \widehat{s_1} \Rightarrow (\hat{s}, \widehat{s_1}) \in \widehat{R}$.
- $a \in \widehat{L}(\hat{s}) \Leftrightarrow (\forall s \in S \, . \, \alpha(s) \Rightarrow a \in L(s))$ for every atomic proposition $a \in AP$.

Example 4.3. Let a be an atomic proposition $a \triangleq (x = 1)$, $\mathcal{P}$ be a set of two predicates $p_1 \triangleq (x > 0)$ and $p_2 \triangleq (y = 1)$ and a model M with one initial state $s_0 \triangleq \langle x = 1, y = 0 \rangle$, two other states $s_1 \triangleq \langle x = 2, y = 2 \rangle$, $s_2 \triangleq \langle x = 1, y = 1 \rangle$ and only one transition from s_0 to s_2. In the abstract model, we have:

- Two concrete states s_0, s_1 are mapped to the same abstract state $\widehat{s_0} \triangleq \langle p_1 = \top, p_2 = \bot \rangle$.
- The state s_2 is mapped to an abstract state $\widehat{s_2} \triangleq \langle p_1 = \top, p_2 = \top \rangle$.
- There exists only one transition from $\widehat{s_0}$ to $\widehat{s_2}$. Notice that while s_1 and s_2 are disconnected, their images $\widehat{s_0} = \alpha(s_1)$ and $\widehat{s_2}$ are connected in the abstract model.
- $\widehat{s_2}$ is labelled with a.
- $\widehat{s_0}$ is not labelled with neither a nor $\neg a$. The reason is that s_0 is marked with a but s_1 is marked with $\neg a$.

Predicate abstraction is an famous instance of existential abstraction [GS97]. The abstract model is constructed from a given set of predicates $\mathcal{P} = \{p_1, \ldots, p_n\}$ over program variables $x_1, \ldots, x_m$. Every predicate p_i is represented by a Boolean variable b_i in the abstract program, while the original variables are eliminated. Since the set of predicates $\mathcal{P}$ is finite, the abstract model is always a finite-state system, even though the original system has infinite states.

The *abstraction function* is usually denoted as $\alpha(s) = \langle p_1(s), \ldots, p_n(s) \rangle$ where $p_i(s)$ is the evaluation of the predicate p_i on the state s. The set of *abstract states* $\widehat{S}$ can be defined as $\widehat{S} = \{\hat{s} \mid \exists s \in S \,.\, \alpha(s) = \hat{s}\}$.

The set $\mathcal{P}$ of predicates and the abstraction function introduce an equivalence relation over states and a partition of the state space. More precisely, if two states s_0 and s_1 in S satisfy the same set of predicates in $\mathcal{P}$, they are equivalence and are mapped to the same abstract state $\hat{s}$. The equivalence classes form a partitioning of the state space, each class is characterized by which of n predicates $p_1, \ldots, p_n$ hold and which do not. Then, every abstract state is a representative of an equivalence class and may be formalize by a bit vector of length n.

Let π be the concrete path $\pi = s_0, s_1, \ldots$ The *abstraction of the path* π is denoted as

$$\alpha(\pi) = \alpha(s_0), \alpha(s_1), \ldots$$

We now can state two main critical properties of existential abstraction

Lemma 4.4 (Over-approximation)**.** Let $\widehat{M}$ be an existential abstraction of M. The abstraction of every path (trace) π in M is a path (trace) in $\widehat{M}$.

$$\pi \in M \Rightarrow \alpha(\pi) \in \widehat{M}$$

Lemma 4.5 (Conservative Abstraction)**.** Let φ be a LTL formula such that the negation operator $(\neg)$ is only applied to the atomic propositions, and let $\widehat{M}$ be an existential abstraction of M. We have that if φ holds on $\widehat{M}$, then it also holds on M.

$$\widehat{M} \models \varphi \Rightarrow M \models \varphi$$

Because of the condition of the labeling function, existential abstraction requires that the set AP must contains all propositions in the formula φ which represents the desired property. If not, we cannot evaluate successful φ on every abstract state.

Existential abstraction is a conservative over-approximative technique. The use of a conservative abstraction generates noticeable reductions in the state space and therefore model checking on the resulting model may be easier than on the original one [1] The weakness of the conservative abstraction is that when model checking of the abstract model fails it may produce a counterexample that does not correspond to a concrete counterexample because of the too coarse abstraction. This counterexample is usually called a *spurious* counterexample. When a spurious counterexample is found, the set of predicates must be adjusted in order to eliminate this counterexample.

A typical way to modify the set of predicates is to add new predicates and this method is named *refinement*. In the past, the model checker needed the user assistance to compute a more sufficiently precise abstraction. Recently, the abstraction refinement process has been automated by the Counterexample Guided Abstraction Refinement (CEGAR) paradigm [BR01, CGJ$^+$00]. While refinement is a growing research area in two last decades, we here do not focus on it. The main reason is that since every TLA$^+$ specification is a logical program, it is not difficult to find a “good” set of predicates. In our work, we assume that $\mathcal{P}$ is given by the user. Moreover, the user has to refine $\mathcal{P}$ manually if needed.

4.3 Model Checking with NuSMV2

Our first attempt was to verify the reduced model generated by predicate abstraction with the model checker NuSMV2 [CCG$^+$02]. NuSMV is a reimplementation and extension of SMV, the first model checker based on BDDs [McM93, CCGR99]. NuSMV2 supports model checking techniques based on binary decision diagrams and propositional satisfiability.

The serious shortcoming of this approach is the full unrolling of the transition relation. Our experiments showed that the unrolling may need more time than checking directly the original model with TLC in some cases. The main reason is that after the translation to the SMT-LIB format, the next predicate is really a complex formula which may contain hundred lines, involving user-defined types, uninterpreted functions and axioms for sets (and functions). Therefore, Z3 spends a lot of time evaluating predicates on it.

In addition, NuSMV2 spends all of its resources in computing one inductive assertion during the running time. This inductive assertion is usually stronger than a desired property. This strategy is refered as *monolithic* [Bra11].

[1] To construct an abstract model, we need spend time evaluating predicates on concrete states and this task may require a lot of time.

Many model checking algorithms have the same problem since they need to compute pre- or post-images precisely or approximately and this computation step also demands many SMT calls. Therefore, we decided not to do experiments with those which require the unrolling of the transition relations.

4.4 Model Checking with IC3

After considerable discussion, we found that IC3 [2] might solve our problem. IC3 is a model checking algorithm for safety properties of a finite state transition system. The main difference between IC3 and other model checking algorithms is that instead of applying a monolithic strategy, it pursues an incremental one. To make a formal verification, IC3 produces lemmas in CNF that are inductive relative to previous lemmas, the safety property and step-wise assumptions [Bra11].

Before explaining how IC3 works with predicate abstraction in detail, we recall some common definitions which are used in the rest of this section.

4.4.1 Background

By x^i, we denote an i-th copy of x and we distinguish the copies by the number of primes. By F^i, we denote a new formula obtained by adding i primes to a formula F. By S^i, we denote a new set obtained by adding i primes to every formula in S. For brevity, we sometimes write x', F' and S' instead of x^1, F^1 and S^1, respectively.

A finite-state transition system is a tuple $S = (X, I, T)$ where X is a set of internal state variables, I is a set of initial states represented by the logical formula $I(\bar{x})$, and a transition relation $T(X, X')$. In our work, S is generated by predicate abstraction. In other words, X is the set $\mathcal{P}$ of predicates. In the rest of this section, by $\widehat{S} = \left(\mathcal{P}, \widehat{I}, \widehat{T}\right)$, we denote the finite-state transition system generated by predicate abstraction and minimal existential abstraction. Notice that the IC3 algorithm does not compute $\widehat{I}$ and $\widehat{T}$.

A state s of a transition system [3] is an assignment of Boolean values to all variables in $\mathcal{P}$ and is represented as a conjunction of literals. Note that the negation of an assignment, denoted by $\neg s$, can be transformed into an equivalent clause. An *inducetive generalization* of a clause d is a (sub-) clause c of s such that all literals of c must appear in s and $c \Rightarrow s$.

An *inductive assertion* for a transition system is a formula F which satisfies two conditions $I \Rightarrow F$ and $F \wedge T \Rightarrow F'$. A formula F is an *inductive strengthening* of a safety property P if $F \wedge P$ is inductive. A formula F is *inductive relative* to another formula G if both $I \Rightarrow F$ and $G \wedge T \wedge F \Rightarrow F'$ hold. Since the assertion G reduces the set of states that

[2]IC3 is an abbreviation of Incremental Construction of Inductive Clauses for Indubitable Correctness
[3]It is generated by predicate abstraction

must be considered, an assertion F may not be inductive on its own (or $F \wedge T \not\Rightarrow F'$). A *Counterexample To Induction* (CTI) is a state which is a bad state or can lead to a bad state.

4.4.2 How IC3 works

IC3 incrementally extends and refines a sequence of frames $\mathcal{F} = F_0, F_1, F_2, \dots, F_k$ that are over-approximations of the sets of states reachable in at most $0, 1, 2, \dots, k$ steps. Frames are represented as logical formulas. Moreover, IC3 requires that all frames must satisfy four following conditions:

(P_1) $I \Rightarrow F_0$,

(P_2) $F_i \Rightarrow F_{i+1}$ for $0 \leq i \leq k$,

(P_3) $F_i \Rightarrow P$ for $0 \leq i \leq k$, and

(P_4) $F_i \wedge T \Rightarrow F'_{i+1}$ for $0 \leq i \leq k$.

The following lemma shows the main property of $\mathcal{F}$

Lemma 4.6. If F_i and F_{i+1} are equivalent for some i, then these above properties imply that F_i is an inductive invariant and $F_i \Rightarrow P$. Therefore, P is an invariant [Bra11].

Proof. First, F_i is closed under the transition relation because of (P_4) and $F_i \Leftrightarrow F_{i+1}$. Second, F_i contains all initial states since (P_1) and (P_2). Thus, F_i is inductive. By (P_3), we have that P is an invariant. Moreover, F_i shows an inductive strengthening of P. □

We now turn to the details of the algorithm. Here, we follow the tutorial by Sebastian Wolff [Wol14]. Algo 4.1 shows the top-level function `prove` which returns **true** if and only if P is S-invariant.

First of all, the satisfiability of $I \wedge \neg P$ and $I \wedge T \wedge \neg P'$ are checked to detect 0- and 1-step counterexamples. This step ensures that all initial state are "good" state and cannot reach a bad state in one step. Otherwise, we have a concrete CTI and the execution stops.

After the primary checks, the sequence $\mathcal{F}$ is initialized to $F_0 = I, F_1 = P$ and the counter is set to $k = 1$. Note that the initialization of $\mathcal{F}$ satisfies all properties (P_1) – (P_4).

On each iteration, first a new frame F_{k+1} is added to the sequence $\mathcal{F}$ (line 28). Second, `prove` calls `strengthen`(k) to strengthen F_i for $1 \leq i \leq k$ so that a bad state cannot be reached in at most k steps. If so, `prove` calls `propagateClauses`(k) to propagate clauses forward through $F_1, F_2, \dots, F_{k+1}$. Finally, if this propagation yields any adjacent frames F_i and F_{i+1} that share all clauses, i.e. $F_i \Leftrightarrow F_{i+1}$, then F_i is an inductive invariant, proving that P is invariant.

Algorithm 4.1: IC3s top-level function `prove()`

```
   input  : A transition system $S$ and a property $P$
   output : A Boolean value
21 if sat($I \land \neg P$) or sat($I \land T \land \neg P'$) then
22     return false
23 end
24 $F_0 \coloneqq I$, clause$(F_0) \coloneqq \varnothing$ ;
25 $F_1 \coloneqq P$, clause$(F_1) \coloneqq \varnothing$ ;
26 $k \coloneqq 1$ ;
27 while true do
28     $F_{k+1} = P$ ;
29     if not strengthen($k$) then
30         return false ;
31     end
32     propagateClauses($k$) ;
33     if $F_i \Leftrightarrow F_{i+1}$ for some $0 \leq i \leq k$ then
34         return true ;
35     end
36     $k \coloneqq k + 1$ ;
37 end
```

In addition, this check with properties of `strengthen`(k) guarantees that the loop at line 27 is always terminated. By P_2, the state sets represented by $F_0, F_1, \dots, F_k$ are non-decreasing with level of k. If the algorithm does not terminate at the final check at line 33, the state sets represented by $F_0, F_1, \dots, F_k$ must be strictly increasing with level of k. Since the number of states is bounded by $max_{no} = 2^{|X|} + 1$, the sequence $\mathcal{F}$ has at most max_{no} elements. Therefore, the loop at line 27 is always terminated.

Algorithm 4.2: IC3s CTI detection `strengthen`

```
   input  : The index of a frame $k$
   output : A Boolean value
38 while sat($F_k \land T \land \neg P'$) do
39     $s \coloneqq$ predecessor of $\neg P'$ extracted from the witness ;
40     if not removeCTI($\langle s, k \rangle$) then
41         return false ;
42     end
43     return true ;
44 end
```

Here, `(un)sat` are calls to the SMT solver. For level k, `strengthen`(k) iterates until F_k removes all states that lead to a bad state in one step. Assume s is one such state.

`strengthen`(k) calls `removeCTI`(s) in 4.3 to eliminate s. The removal keeps all properties P_1 – P_4 true.

Algorithm 4.3: IC3s CTI elimination

```
input  : A bad entry $\langle s,k\rangle$
output : A Boolean value with an explanation for an error trace, if exists
45 $states \coloneqq \{\langle s,k\rangle\}$ ;
46 while $states \neq \varnothing$ do
47     $\langle q,i\rangle \coloneqq$ pop an element of $states$ that minimizes $i$ ;
48     if sat($F_0 \wedge T \wedge \neg q \wedge q'$) then
49         print Counterexample ;
50         return false ;
51     end
       // here, $\neg q$ is at lest inductive relative to $F_0$
52     $j \coloneqq$ maximal $j$ with unsat($F_j \wedge T \wedge \neg q \wedge q'$) ;
53     $c \coloneqq$ inductiveGeneralize($\neg q$) ;
54     for $l \coloneqq 0$ to $j+1$ do
55         $F_l \coloneqq F_l \wedge c$ ;
56     end
57     if $j \leq j-1$ then
58         return true // proof obligation fulfilled
59     end
60     $w \coloneqq$ witness for sat($F_{j+1} \wedge T \wedge \neg q \wedge q'$) ;
61     $p \coloneqq$ predecessor of $q$ extracted from $w$ ;
62     $states \coloneqq states \cup \{\langle p,j+1\rangle\}$ ;
       // $\langle p,j+1\rangle$ might not resolve $\langle q,i\rangle \rightsquigarrow$ check again
63     $states \coloneqq states \cup \{\langle q,i\rangle\}$ ;
64 end
65 return true ;
```

IC3 maintains a whole set of such proof obligations. Each entry $\langle q,i\rangle$ in *states* mentions that a state q must be eliminated from frame (formula) F_i. To do so, we prove that (1) F_{i-1} implies $\neg q$ and (2) $\neg q \wedge F_{i-1} \wedge T \wedge q'$ is unsatisfiable. By (1) and (P_2), we have that $F_l \Rightarrow \neg q$ for $0 \leq l \leq i$ and therefore we can conjoin a clause c to frames $F_0, \ldots, F_i$ where c is a inductive generalization of $\neg q$, i.e. $c \Rightarrow \neg q$. By (2), we have that q is removed from F_i. Finally, from (1) and (2) we can conclude that $\neg q$ is inductive relative to F_{i-1}. It means that F_i implies $\neg q$.

Let us now explain how the function `removeCTI`(s) in 4.3 generates correct proof obligations for $\langle s,k\rangle$ (and $\langle q,i\rangle$).

First of all, `removeCTI` checks if q is reachable from F_0 just in one step (line 48) [4]. If so, we can construct a counterexample and the algorithm stops. Otherwise, we seek a maximal j such that $F_j \wedge T \wedge \neg q \wedge q'$ is unsatisfiable. Due to the first check at line 21, j must exist. We also have that $j \geq i-2$. If not, $\langle q, i\rangle$ can be reached in at most $i-1$ steps. Contradiction. Moreover, we can prove that $\neg q$ is inductive relative to each frame F_l for $0 \leq l \leq j$.

Next, we can conjoin an inductive generalization c of $\neg q$ to $F_0, \ldots, F_{j+1}$. Certainly, we can choose $c = \neg q$, but this might be inefficient because only the single state q is removed. The generalization can eliminate not only q but also other $q - like$ states. However, instead of the algorithm `down` in [Bra11], we choose c is an unshifted unsatisfiable core of $F_j \wedge T \wedge q'$ such that every literal of c must be a literal of q'. If $j \geq i-1$, we are done. Otherwise, we know that a lemma is not enough "strong" and we need to find a better one.

Assume that $j < i-1$. Hence, we have a witness w of $F_{j+1} \wedge T \wedge \neg q \wedge q'$ (line 60). Let p be a predecessor of q extracted from w. Now our additional minor goal is to generate a proof obligation of p since q is reachable through p. Therefore, we add $\langle p, j+1\rangle$ to *states* (line 62).

Eventually, obligation $\langle p, j+1\rangle$ fulfills and p is eliminated from F_{j+1}. Then $\langle q, i\rangle$ is considered again and the procedure is rerun. However, the query $F_{j+1} \wedge T \wedge \neg q \wedge q'$ will do not have a witness with p again because of the previous strengthening. Hence, a new predecessor of q is revealed and is treated like p. In general, we may need to find a proof obligation for every predecessor of q. Since the number of states in finite, `removeCTI` always terminates.

Notice that properties $(P_1) - (P_4)$ in $\mathcal{F}$ are maintains during `removeCTI`.

(P_1) No initial state is eliminated from F_0. Since F_0 implies $\neg q$ (enforced by check at line 21 and 48), the conjunction of F_0 and c does not change the set represented by F_0. Indeed, c is actually inductive relative to F_0.

(P_2) Clauses are always added to all frames $F_0, \ldots, F_i$ for some i and therefore, we have $F_i \Rightarrow F_{i+1}$.

(P_3) We only add more clauses to frames.

(P_4) A state q and all its predecessors are eliminated, together.

Finally, we need to propagate clauses forward to refine frames. That is, some clause c of frame F_i is add to frame F_{i+1} (line 69). To do that, we need to check that $F_i \wedge T \wedge \neg c'$ is unsatisfiable. That is, F_{i+1} does not contain $\neg c$ and $\neg c$ is unreachable in at least $i+1$ steps. Here, we focus on only one clause. However, in theory, this is not the strongest possible refinement. The combination of two or more clauses may be inductive relative to F_i but no clause is.

[4]Notice that the initial check at line 21 guarantees that F_0 does not contain q.

Algorithm 4.4: IC3s clause pushing `propagateClauses`

```
input  : The number of frame $k + 1$
66 for $i \coloneqq 1$ to $k$ do
67     for each clause $c \in clause(F_i)$ do
68         $F_i \coloneqq F_i \wedge c$ ;
69         clause$(F_i) \coloneqq clause(F_i) \cup \{c\}$ ;
70     end
71 end
```

Finally, the following lemma shows the correctness and termination of IC3. This lemma can be proved based on ideas we discuss in the previous paragraphs.

Lemma 4.7. Given a finite state transition system S and a formula P, IC3 terminates and returns **true** if and only if P is a safety property of S [Bra11].

IC3 with predicate abstraction It is easy to modify the above algorithms to combine IC3 with predicate abstraction. Literals now show the truth value of predicats and frames are over-approximations of abstract states. Moreover, instead of a concrete system and SAT calls, IC3 now is applied on an abstract system and calls a SMT solvers. For example, a query at line 52 is replaced by `(un)sat`$_{\texttt{SMT}}$ $(F_j \wedge T \wedge \neg\hat{q} \wedge q')$.

Spurious counterexamples can happen because of two following reasons [BBW14]:

1. An abstract state p represents both reachable and unreachable states and the unreachable subset of $\hat{p}$ leads to an abstract CTI $\hat{q}$. Since $\hat{p}$ covers reachable states, IC3 cannot remove it and erroneously concludes that a counterexample is discovered.

2. $F_i \wedge T \wedge \neg\hat{s} \wedge \widehat{s'}$ is satisfiable while $F_i \wedge T \wedge \neg s \wedge s'$ is not. The reason is that an abstract state $\hat{s}$ resembles some concrete states that are reachable in at most $i - 1$ steps and concrete states that are reachable in at least i steps.

While the first scenario is a common challenge in predicate abstraction, the second one is a typical problem of IC3. Therefore, when reviewing a counterexample, the user needs to check whether concrete states characterized by an abstract state are reachable in the same number of steps. Notice that our system has not supported automatic refinement and the user needs to change a set of predicates manually.

Since the lack of automatic refinement, the implementation of IC3 with predicate abstraction is similar with the original one, except that we now consider only abstract states $\hat{q}$ (not concrete states q). Notice that we still use the original initial and transitional formulas.

4.5 Conclusions

In this chapter, first we have introduced model checking and its features. In addition, we have described how to copy with the state-space explosion problem by predicate abstraction. We have showed some results of our preliminary experiments with NuSMV2 and given reasons why monolithic is not an appropriate strategy for our model checker. Finally, we have explained how to generate an incremental proof for a safety property with the IC3 algorithm. In our preliminary experiments, IC3 could verify safety properties in acceptable time since it did not need to unroll the transition translation. Therefore, we decided to apply the IC3 algorithm. Our model checker is the integration of IC3 and predicate abstraction.

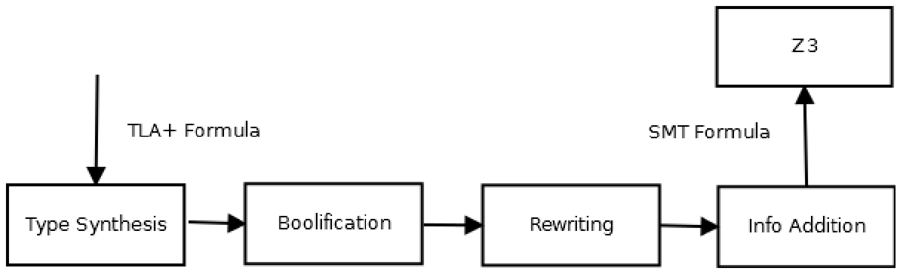
CHAPTER 5

Translation of TLA$^+$ to Z3

5.1 Overview

The main goal of this section is to develop an efficient translation from the fragment of TLA$^+$ discussed in Section 5.2 to the language of the SMT solver Z3. The resulting Z3 specification should be efficient enough to construct a predicate abstraction which is used to check the invariant. The translation proceeds in four main steps: type synthesis, Boolification, expression rewriting and information addition which contains symbol declarations and assertions of axioms.

Figure 5.1: Our translation process



Our framework for the SMT-LIB translation is presented in Fig. 5.1. In general, the translation has four main steps. First, based on the type-correctness invariant which is given by users in order to know the possible values of variables, our system constructs and assigns type information to every expression in a TLA$^+$ specification. Our system will stop and give the user an error message if it finds a type violation. Second, the Boolification procedure performs a quick double-check of expressions' types. Then, our tool rewrites TLA$^+$ formulas that Z3 cannot handle directly. At that point, most expressions have a native counterpart in the language of Z3. Finally, extra information for symbol declarations and assertions of axioms is added. Now, the it is ready to pass the result to the solver.

Given a solver's input language, such as the SMT-LIB extension of Z3, we call a *basic* formula one that is formed by TLA$^+$ expressions that can be directly written in that target language. In other words, a basic TLA$^+$ expression has a counterpart in the target language. Boolean operators and most of operators defined in the module *Integers* are one-to-one corresponding with the predefined operators of SMT-LIB/AUFLIA. However, operators and constructs for sets and functions are not. Therefore, we need to rewrite these expressions into those which can be straightforwardly passed to the SMT solver.

Researchers have recently made attempts on translating non-temporal TLA$^+$ part into many-sorted (first-order) logic. Hansen and Leuschel introduced a framework to translate TLA$^+$ to B for validation with ProB [HL12, LB03]. However, ProB is an explicit model checking tool and it does not offer advatages of predicate abstraction. Constructing a predicate abstraction for a B specification is not our focus. Merz and Vanzetto presented approaches to encode TLA$^+$ proof obligations into many-sorted logic and to integrate automated theorem provers and SMT solvers into TLAPS. First, they tried to encode sets by characteristic predicates but we cannot represent a set of sets or functions with characteristic predicates [MV12]. Therefore, this method does not meet our requirements. Later, they suggest the single-sorted encoding for TLA$^+$, which is also called the untyped encoding. This approach can handle a useful fragment of the TLA$^+$ language, including set theory, functions, linear arithmetic expressions and especially the CHOOSE operator (Hilbert's choice) [MV12]. The main weakness of the untyped encoding is that this mechanism introduces many additional quantifiers and defines many "fresh" relations, even for built-in operators in SMT-LIB what we say the implications of having many quantifiers. In order to reduce the number of quantifiers and to utilize features in SMT solvers, Merz and Vanzetto proposed a TLA$^+$ type system using refinement and dependent types [MV14]. However, deciding a refinement and dependent type for a TLA$^+$ expression is an undecidable problem. Moreover, if their typed system cannot decide an appropriate type for an expression, such as the empty set, they will come back to the untyped encoding. Therefore, their systems are not efficient enough to reason about a next-state action with an SMT solver since a proof obligation is usually more "shallow" than the nsext predicate.

First, we introduce our new TLA$^+$ fragment TLA$^{+\mathcal{T}}$ which is smaller than one which TLC can handle, but we believe our fragment is still expressive enough to write specifications for many distributed algorithms. Our TLA$^+$ fragment requires an over-approximation on a function and accepts most relevant expressions records, tuples and set operators. Some unsupported features on our fragment are the CHOOSE operator, an empty sequence, and a set of elements with different types. In contrast to the standard TLA$^+$, expressions in our fragment are distinguished into different syntactic categories: terms t, formulas φ, numbers n, strings str, sets s, functions fcn, records rcd, tuples tup and expressions e.

Second, our type system $\mathcal{T}$ is presented. In general, each kind of syntactic categories except terms has a one-to-one corresponding class of types. SMT basic types are

Bool, *Int*, and *String* and constructed types are classified into four categories which are sets, functions, records and tuples. In addition, our type system contains type variables α which can be thought of as unknown types and are interpreted over the resulting Herbrand universe generated by type constructors. A set of types in a given TLA$^+$ specification is constructed based on the TLA$^+$ invariant *TypeOK*.

Before the translation, we need to check whether or not each expressions e in a given TLA$^+$ specification *Spec* can be assigned a suitable type. To do that, we implement a light constraint-based type-checking procedure which has two main modules: a unification one and a constraint one. The main purpose of the first module is to construct a type assignment σ based on type inference rules which are purely syntactic. After that, the second module generates corresponding constraints which must be satisfied by the assignment σ. A TLA$^+$ expression is $\mathcal{T}$ well-formed if and only if our type checker can assign it (and its sub-expressions) a type. Notice that to avoid conflicts, all (bounded) variables should be renamed so that no two (bounded) variables have the same name in order to avoid ambiguity. The renaming procedure can be done automatically.

Because TLA$^+$ is an untyped language, there is no syntactic differences between TLA$^+$ Boolean expressions and non-Boolean ones. In [Van14], Vanzetto introduces an algorithm to distinguish them. We extend Vanzetto's algorithm and use the extension to perform a quick double-check of expressions' types. Our algorithm is complete for TLA$^{+\mathcal{T}}$.

If no type errors are found, the translation starts. Hence, rewriting rules are applied to transform a complex TLA$^+$ expression to a simpler one. All rewriting rules ensures that a resulting formula is $\mathcal{T}$ well-formed. Moreover, many heuristics are applied during the transformation period to simplify the resulting expression and to improve the performance of the SMT solver. The translation stops when all expressions are in the basic form.

Finally, a shallow embedding is added. This step adds extensionality axioms for sets and functions. Moreover, it declares useful constants and constraints for strings. The resulting specification now can pass directly to Z3.

5.2 The fragment of TLA$^+$

In this subsection, we define a fragment of the TLA$^+$ language, which will be used in the rest of this chapter and be called TLA$^{+\mathcal{T}}$ [1]. The fragment considered here is smaller than one which TLC can handle, but we believe our fragment is expressive enough to write specifications for many distributed algorithms. In contrast to the standard TLA$^+$, expressions in our fragment are distinguished into different syntactic categories: terms t, formulas φ, numbers n, strings *str*, sets s, functions f, records *rcd*, tuples *tup*, and expressions e. Moreover, TLA$^{+\mathcal{T}}$ requires the type invariant *TypeOK* in every specification and has some restrictions which are mentioned later.

[1]The symbol TLA$^{+\mathcal{T}}$ implies that this fragment will be decorated with the type system described in the next section.

Restrictions in TLA$^{+\tau}$ are

- The CHOOSE operator, the empty sequence, the general construct $\{e : y_1 \in S_1, \ldots, y_n \in S_n\}$, recursive definitions, a function application with many arguments are not allowed.
- Temporal operators are not considered.
- The CASE construct without OTHER is also removed.
- Every expression in TLA$^{+\tau}$ must have a TLC value [2].
- After the type synthesis, all elements of a set must have the same type [3].
- All elements in the domain of a function f should have the same type in our type system which is described in detail later.
- The codomain of a function f, which is $\{f[x] : x \in \mathsf{DOMAIN}\, f\}$, should be a TLC value and all of its elements should have the same type. For example, a function

  $$[x \in \{1, 2\} \mapsto \mathsf{IF}\ x = 1\ \mathsf{THEN}\ f[x] = 0\ \mathsf{ELSE}\ f[x] = \text{“a”}]$$

  is illegal.
- A TLA$^{+\tau}$ string is considered as a constant, not a sequence of characters [4].

In the following, we use notations

- x is a variable,
- c is a constant which can be TRUE, an integer or a tuple constant,
- $s, \mathit{fcn}, \mathit{rdc}, \mathit{tup}$ are respectively abbreviations for sets, functions, records and tuple, and
- $\otimes, \oplus, \circ$ are a Boolean operator, an arithmetic comparison operator, a set operator, respectively.

TLA$^{+\tau}$ is defined with the following grammar

TLA$^+$ is an untyped system, and therefore type-correctness is not imposed by the language. However, every specification in TLA$^{+\tau}$ must have the type invariant for its variables and ensure that all quantified variables are also bound to a finite set. For instance, both of two expressions $\forall x.\varphi$ and $\forall x \in \mathit{Int}.\varphi$ are discarded.

Finally, it is use to see that TLA$^{+\tau}$ is a subset of a fragment on which TLC can work. An expression satisfying all above requirements is called a TLA$^{+\tau}$ well-formed expression.

[2]This requirement implies that an argument of a function application is always in the function domain. When we rewrite a function application, this condition is automatically added.

[3]This requirement implies restrictions for function's domain and codomain.

[4]However, we do not think it is a weakness of TLA$^{+\tau}$ since in practice, the user seldom uses a string as a sequence.

$(\textit{terms})$	$t ::= x \mid c \mid \mathit{fcn}[e] \mid \mathit{rcd}.h$
$(\textit{formulas})$	$\varphi ::= t \mid \mathsf{FALSE} \mid \mathsf{TRUE} \mid \varphi \otimes \varphi \mid e = e \mid e \oplus e \mid e \in e \mid e \subseteq e \mid \forall x : \varphi \mid \exists x : \varphi$
$(\textit{integers})$	$n ::= t \mid \ldots \mid -1 \mid 0 \mid 1 \mid \ldots \mid n + n \mid n - n \mid n \star n \mid n \text{ div } n \mid n \text{ mod } n$
$(\textit{strings})$	$\mathit{str} ::= t \mid \text{“abc”}$
$(\textit{sets})$	$s ::= t \mid \{\} \mid \mathit{BOOLEAN} \mid \mathit{Int} \mid s \circ s \mid$ $\{e, \ldots, e\} \mid \mathsf{SUBSET}\, s \mid \mathsf{UNION}\, s \mid \{x \in s : \varphi\} \mid \{e(x) : x \in s\} \mid$ $[s \to s] \mid [h_i : s_i] \mid s \times \ldots \times s \mid$ $\mathsf{DOMAIN}\, \mathit{fcn} \mid \mathsf{DOMAIN}\, \mathit{rcd} \mid \mathsf{DOMAIN}\, \mathit{tup}$
$(\textit{functions})$	$\mathit{fcn} ::= t \mid [x \in s \mapsto e]$
$(\textit{records})$	$\mathit{rcd} ::= t \mid [h_i \mapsto e_i]$
$(\textit{tuples})$	$\mathit{tup} ::= t \mid \langle e, \ldots, e \rangle$
$(\textit{expressions})$	$e ::= t \mid \varphi \mid n \mid \mathit{str} \mid s \mid \mathit{fcn} \mid \mathit{rcd} \mid \mathit{tup}$

5.3 Our type system

5.3.1 Syntax and definition

In principle, types τ for the fragment TLA$^{+\tau}$ are defined by the following grammar

$$\begin{aligned} \tau ::=& \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Str} \mid \\ & \mathsf{Set}\, \tau \mid \tau \to \tau \mid [h \mapsto \tau] \mid \langle \tau \rangle \mid \alpha \end{aligned}$$

Bool, Int, and Str are atomic types for Boolean formulas, integers and strings, respectively. The type constructor Set determines the level of set strata, for instance, for the enumeration or the operators SUBSET and UNION. The type construct $\tau \to \tau$ is corresponds to a unary function. The type construct $[h \mapsto \tau]$ and $\langle \tau \rangle$ are for records and tuples. Type variables α, representing unknown types, are interpreted over the resulting Herbrand universe induced by the preceding type constructors, that is, the set of all ground, i.e. variable-free, types. We usually note atomic types by the letter β, and ground types by the letter γ. A ground assignment σ is a mapping, maybe partial, of type variables to ground types, where $\square$ is the empty assignment.

$$\sigma ::= \square \mid \alpha \mapsto \gamma, \sigma$$

Example 5.1. Both sets $S_1 = \{1, 2\}$ and $S_2 = \{3, 2\}$ have the same type $\mathsf{Set\ Int}$. A variable z in an expression $z \in S_1$ has the type Int. Both functions $[x \in S_1 \mapsto 0]$ and $[x \in S_2 \mapsto 0]$ have the same type $\mathsf{Int} \mapsto \mathsf{Int}$. Two sets of functions $[S_1 \to S_1]$ and $[S_2 \to S_1]$ have the same type $\mathsf{Set}\ (\mathsf{Int} \mapsto \mathsf{Int})$. A type of a record $[\text{“rnd”} \mapsto 1, \text{“val”} \mapsto \text{“ECHO”}]$ is $[\text{“rnd”} : \mathsf{Int}, \text{“val”} : \mathit{Str}]$. A type of a tuple $\langle 1, \mathsf{TRUE} \rangle$ is $\langle \mathsf{Int}, \mathsf{Bool} \rangle$.

In the following, we give some ideas how our type system works through small examples.

The empty set

The empty set $\{\}$ is the set having no elements. In the set theory ZFC, the empty set unique and is represented as $\forall x . x \notin \{\}$. However, in a many-sorted type system, it is natural to think that there is an empty set in every type for sets. In our type system, many empty set exist.

The following examples show some challenges when reasoning about type information for empty sets.

Example 5.2. A type of an expression, especially the empty set, depends on related expression in some cases. Consider an expression $e_2 \triangleq \{\} = \{1\}$. It is easy to see that e_2 should the type Bool since it is an equation. However, to decide a type of the empty set at the left-hand side of e, we need to look at the right-hand side. Since $\{1\}$ has the type $\mathsf{Set\ Int}$, the empty set here should have the same type $\mathsf{Set\ Int}$.

Example 5.3. In some cases, it is not easy to know a type of an expression, including an empty set. For example, in an expression $e_3 \triangleq \{\{\}\} \cup \{\{\text{“a”}\}\}$, the set $\{\{\}\}$ has the type $\mathsf{Set\ Set\ Str}$ and its inner one has the type $\mathsf{Set\ Str}$.

Example 5.2 and 5.3 show that the empty set can be assigned different types. Because of the type discipline, we need to discard some TLA$^+$ well-formed expressions with the empty set. For example, an expression $e_4 \triangleq \{1\} \setminus \{1\} = \{\text{“a”}\} \setminus \{\text{“a”}\}$ is evaluated as TRUE under all of TLA$^+$ interpretations which are the conservative, liberal and moderate interpretations. However, in our system, e_4 is a ill-formed expression since its left-hand side and right-hand side have different types, $\mathsf{Set\ Int}$ and $\mathsf{Set\ Str}$.

Sets

All elements of a set should have the same type; otherwise, a set has an undefined type. The main advantage of this requirement is that it makes our type system simple. In order to represent a set of elements with different types, we need to define an hierarchy of types. At the moment, SMT solvers do not supported subtype declarations which makes the reasoning procedure complex. Therefore, the set $s \triangleq \{1, \text{“a”}\}$ is ill-formed in our fragment. Fortunately, we can encode many distributed algorithms in the way no set of elements with different types are used. For example, the user can read our encodings of Chandra's algorithm for reliable broadcast by message diffusion (BcastFolklore) [CT96] and Raynal's algorithm for non-blocking atomic commitment (NBAC) [Ray97]. However, this restriction is really a weakness of our type system. For instance, our system cannot handle expressions in the TLA$^+$ Paxos specification of Leslie Lamport because a variable *bmsgs* is a set of records with different structures [Lam11].

Functions

Many functions which TLC can evaluate are assigned undefined types in our type system. Assume that $s_1 \triangleq \{1, 2\}, s_2 \triangleq \{\text{“a”}\}, S \triangleq \{s_1, s_2\}$, and $f \triangleq [x \in S \mapsto 0]$. Since cardinalities of s_1 and s_2 are not equal, s_1 and s_2 are TLA$^+$ comparable. Hence, S has a TLC value and f can be evaluated by TLC. However, s_1 and s_2 have different types: s_1 has type $\mathsf{Set\ Int}$ and s_2 has type $\mathsf{Set\ Str}$. Therefore, types of S and f are undefined. In general, the type construct $\tau \rightarrow \tau$ cannot capture functions whose domain (and codomain) has elements with different types.

Moreover, the over-approximation on functions can make an invalid TLA$^+$ formula become valid. Assume that a function f is declared as $f_1 \triangleq [x \in \{1, 2\} \mapsto 1]$. Now consider the formula $\varphi \triangleq \forall x \in \mathsf{Int}. f_1[x] + 0 = f_1[x]$. If $x = 3$, the value of $f[3]$ in TLA$^+$ is undefined since 3 is not in the domain of f. Therefore, we cannot compare $f_1[3] + 0$ with $f_1[3]$. In other words, φ is invalid in TLA$^+$. However, in our type system, φ becomes valid. First, f is assigned the type $\mathsf{Int} \rightarrow \mathsf{Int}$. Although we do not still know the value of $f[3]$, now we ensure that its value is an integer. Therefore, $f_1[3] + 0 = f_1[3]$. In general, for all $x \in Int$, we have $f_1[x] + 0 = f_1[x]$. It means φ is valid in our type system. To avoid this kind of errors, the user should check whether an argument of a function application is in the function domain. In our rewriting system, this condition $x \in \mathsf{dom}(f)$ is automatically added.

Notice that the formula $\varphi' \triangleq \forall x \in \mathsf{Int}. x \in \mathsf{dom} \Leftrightarrow (f_1) f_1[x] + 0 = f_1[x]$ is valid in both untyped and typed versions of TLA$^+$.

Functions, records and tuples

In contrast to the standard version of TLA$^+$, here functions, records and tuples are classified into three different categories and have different types. The main reasons are the over-approximation on functions and the operator DOMAIN .

- The type construct $\tau \rightarrow \tau$ requires that all elements in the codomain of a function f must have the same type. If a tuple *tup* considered as a function, then the type of *tup* has to be necessarily compatible with the type of a TLA$^+$ function. It means *tup* has a functional type $\tau_1 \rightarrow \tau_2$ or all elements of *tup* should be of the same type. As a result, we cannot use a tuple with elements of different types, i.e. $\langle 1, \mathsf{TRUE} \rangle$. Therefore, to enhance the power of our type system, we decide to reorganise functions and tuples into different classes.
- Functions and records are separated because of the similar reason.
- In TLA$^+$, a record *rcd* is a function whose domain is a set of strings and a tuple *tup* is a function whose domain is set of integers. It means their domains DOMAIN *rcd* and DOMAIN *tup* are always assigned different types. Therefore, it makes sense to have distinguished type constructs for records and tuples.

The categorization of functions, records and tuples causes some troubles. Let's consider a record $r \triangleq [1 \mapsto 1, 2 \mapsto 1]$ and a function $f \triangleq [x \in \{1,2\} \mapsto 1]$. In TLA$^+$, r equals f. However, r and f are incomparable in our type system because they are assigned different types. We do not see this as a strong limitation as our type constructor will complin to the user about type inconsistency, and the user can fix the specification accordingly. Notice that any type system requires discipline. In general, if a record appears in a given TLA$^+$ specification explicitly as a function, it will not be captured by our type discipline. To avoid ambiguity, the user should not use the function application for a record.

Special type operators

We introduce four new special type operators in order to extract information from a function, record and tuple type.

$$\tau ::= \ldots \mid \mathsf{dom}\,(\tau) \mid \mathsf{cod}\,(\tau) \mid \mathsf{dot}\,(\tau, h) \mid \mathsf{get}\,(\tau, i)$$

These type modifiers are defined by the properties

$$\begin{array}{ll}
\mathsf{dom}\,(\tau_1 \to \tau_2) = \tau_1 & \mathsf{dom}\,([h_i \mapsto \tau_i]_{i=1..n}) = \mathsf{Str} \\
\mathsf{dom}\,(\langle \tau_1, \ldots, \tau_n \rangle) = \mathsf{Int} & \mathsf{cod}\,(\tau_1 \to \tau_2) = \tau_2 \\
\mathsf{dot}\,([h_i \mapsto \tau_i]_{i=1..n}, h_i) = \tau_i & \mathsf{get}\,(\langle \tau_1, \ldots, \tau_n \rangle, i) = \tau_i
\end{array}$$

The type $\mathsf{dom}\,(\tau)$ represents the domain of some type τ, when τ is a function, record, or tuple type. The type $\mathsf{cod}\,(\tau)$ describes the codomain of some function type τ. The type $\mathsf{dot}\,([h_i \mapsto \tau_i]_{i=1..n}, h_i)$ mentions that we want to know type information related to a record selection at a field h_i. Finally, the type $\mathsf{get}\,(\langle \tau_1, \ldots, \tau_n \rangle, i) = \tau_i$ is some type τ_i at a position i. Notice that two different ground types in $\mathcal{T}$ are never unifiable.

By applying their properties as rewriting rules, the type operators $\mathsf{dom}, \mathsf{cod}, \mathsf{dot}, \mathsf{get}, \omega$ can be eliminated when they are applied to the expected type.

5.3.2 Type relations

A type system usually allows two equivalence relations $\equiv$ and $\cong$ on types, whose difference is that the first one allows type variables to be unified, while the latter is the equality between ground types defined above [MV14].

Two ground types γ_1 and γ_2 are equal, noted $\gamma_1 \cong \gamma_2$, if and only if they characterize the same elements. Note that in our system any two different ground types γ_1 and γ_2 with syntactically different representations are always disjoint. Therefore, we say that $\gamma_1 \cong \gamma_2$ is valid if and only if γ_1 and γ_2 are exactly the same object.

Two types τ_1 and τ_2 are unifiable, noted $\gamma_1 \equiv \gamma_2$, if and only if there exists a type assignment σ that makes $\sigma\tau_1 \cong \sigma\tau_2$. We usually write $\sigma \vDash \tau_1 \equiv \tau_2$.

Moreover, we define a new type relation $\lhd$. we say that $\gamma_1 \lhd \gamma_2$ if and only if γ_1 and γ_2 are are unifiable and γ_1 is a ground type. The type relation $\lhd$ is a special case of the type relation $\cong$ and appears only in inference rules for equality and two set operators $\in$ and $\subseteq$. We use this type relation to ensure that the upper bound of possible values of some (left-hand or right-hand) side is known.

Example 5.4. The unification of the types $\mathsf{Set}\,\alpha_1 \rightarrow \mathsf{Set\,Int} \equiv \mathsf{Set\,Bool} \rightarrow \alpha_2$ yields the ground assignment $\sigma = \alpha_1 \mapsto \mathsf{Bool}, \alpha_2 \mapsto \mathsf{Set\,Int}$. Then, $\sigma\,(\mathsf{Set}\,\alpha_1 \rightarrow \mathsf{Set\,Int}) \cong \sigma\,(\mathsf{Set\,Bool} \rightarrow \alpha_2)$ is valid.

Example 5.5. The unification of the types $\mathsf{Set}\,\alpha_1 \rightarrow \mathsf{Set}\,\alpha_3 \equiv \mathsf{Set\,Bool} \rightarrow \alpha_2$ yields the ground assignment $\sigma_1 = \alpha_1 \mapsto \mathsf{Bool}$. Then, $\sigma_1\,(\mathsf{Set}\,\alpha_1 \rightarrow \mathsf{Set\,Int}) \cong \sigma_1\,(\mathsf{Set\,Bool} \rightarrow \alpha_2)$ is not valid.

Just for presentational purposes, sometimes we write $\tau_1 \equiv \ldots \equiv \tau_n$ as $\tau_i \equiv \tau_j$ for all $i, j \in 1\,..\,n$ and $\tau_1 \cong \ldots \cong \tau_n$ as $\tau_i \cong \tau_j$ for all $i, j \in 1\,..\,n$.

While our type inference rules uses three above type relations to describe constraints, our constraint solvers uses only the non-unifiable relation $\cong$ to generate constraints.

5.3.3 Typing propositions and typing hypotheses

Here, we adapt the definition of the type proposition in [Van14]. The *typing proposition* of a type assignment $e : \tau$, noted $[\![e : \tau]\!]$, is a characteristic predicate associated to the type τ that is true for precisely those expressions e whose possible values are in the type τ. In other words, the TLA$^+$ formula $[\![e : \tau]\!]$ states that the expression e is one of the elements characterized by the type τ. For instance, having an integer type is characterized by being a member of the set of integers. Typing propositions allow us to syntactically translate a type assignment $x : \tau$, where τ is a ground type, to a TLA$^+$ formula $[\![x : \tau]\!]$.

Definition 5.6. Given a TLA$^+$ expression e and a ground type τ, the TLA$^+$ formula $[\![e : \tau]\!]$ is defined as follows:

$$
\begin{array}{rrl}
\text{Boolean} & [\![e : \mathsf{Bool}]\!] & \triangleq e \in \mathit{BOOLEAN} \\
\text{Integer} & [\![e : \mathsf{Int}]\!] & \triangleq e \in \mathit{Int} \\
\text{String} & [\![e : \mathsf{Str}]\!] & \triangleq e = \text{“abc”} \\
\text{Set} & [\![e : \mathsf{Set}\ \tau]\!] & \triangleq \forall x \in e : [\![x : \tau]\!] \\
\text{Set of records} & [\![e : \mathsf{Set}\ [h_i \mapsto \tau_i]_{i:1..n}]\!] & \triangleq \land\, e = [h_1 : e_1, \dots, h_n : e_n] \\
 & & \quad \land\, [\![h_1 : \mathsf{Str}]\!] \land \dots \land [\![h_n : \mathsf{Str}]\!] \\
 & & \quad \land\, [\![e_1 : \mathsf{Set}\ \tau_1]\!] \land \dots [\![e_n : \mathsf{Set}\ \tau_n]\!] \\
\text{Set of tuples} & [\![e : \mathsf{Set}\ \langle \tau_i \rangle_{i:1..n}]\!] & \triangleq \land\, e = e_1 \times \dots \times e_n \\
 & & \quad \land\, [\![1 : \mathsf{Int}]\!] \land \dots \land [\![n : \mathsf{Int}]\!] \\
 & & \quad \land\, [\![e_1 : \mathsf{Set}\ \tau_1]\!] \land \dots [\![e_n : \mathsf{Set}\ \tau_n]\!] \\
\text{Function} & [\![e : \tau_1 \to \tau_2]\!] & \triangleq \land\, e = [x \in \mathsf{DOMAIN}\ e \mapsto e[x]] \\
 & & \quad \land\, \forall x : x \in \mathsf{DOMAIN}\ e \Leftrightarrow [\![x : \tau_1]\!] \\
 & & \quad \land\, \forall x : [\![x : \tau_1]\!] \Rightarrow [\![e[x] : \tau_2]\!] \\
\text{Set of functions} & [\![e : [h_i \mapsto \tau_i]_{i:1..n}]\!] & \triangleq \lor\, (\land\, e \in [h_1 : e_1, \dots, h_n : e_n] \\
 & & \qquad \land\, [\![[h_1 : e_1, \dots, h_n : e_n] : \mathsf{Set}\ [h_i \mapsto \tau_i]_{i:1..n}]\!]) \\
 & & \quad \lor\, (\land\, e = [h_1 \mapsto e_1, \dots, h_n \mapsto e_n] \\
 & & \qquad \land\, \mathsf{DOMAIN}\ e = \{h_1, \dots, h_n\} \\
 & & \qquad \land\, [\![h_1 : \mathsf{Str}]\!] \land \dots \land [\![h_n : \mathsf{Str}]\!] \\
 & & \qquad \land\, [\![e_1 : \tau_1]\!] \land \dots \land [\![e_n : \tau_n]\!])
\end{array}
$$

$$
\begin{array}{rrl}
\text{Tuple} & [\![e : \langle \tau_i \rangle_{i:1..n}]\!] & \triangleq \lor\, (\land\, e \in e_1 \times \dots \times e_n \\
 & & \qquad \land\, [\![e_1 \times \dots \times e_n : \mathsf{Set}\ \langle \tau_i \rangle_{i:1..n}]\!]) \\
 & & \quad \lor\, (\land e = \langle e_1, \dots, e_n \rangle \\
 & & \qquad \land\, \mathsf{DOMAIN}\ e = \{1, \dots, n\} \\
 & & \qquad \land\, [\![1 : \mathsf{Int}]\!] \land \dots \land [\![n : \mathsf{Int}]\!] \\
 & & \qquad \land\, [\![e_1 : \tau_1]\!] \land \dots [\![e_n : \tau_n]\!])
\end{array}
$$

To ensure that types are pairwise disjoint, we should introduce the set of axioms

$$\forall x, y : [\![x : \beta_1]\!] \land [\![y : \beta_2]\!] \to x \neq y$$

for each pair of atomic types $\{\mathsf{Bool}, \mathsf{Int}, \mathsf{Str}\}$.

Definition 5.7. Given a typed-TLA⁺ expression e, the relativized expression $\mathcal{R}(e)$ is obtained by recursively replacing each type annotation $x : \tau$ by a new hypothesis $[\![x : \tau]\!]$.

The relevant transformation rule for a quantified formula is

$$\mathcal{R}(\forall x^{\tau} : \varphi) \triangleq \forall x : [\![x : \tau]\!] \Rightarrow \mathcal{R}(\varphi)$$

As in standard relativization that ensures soundness in the translation from MS-FOL to FOL, we have that the following lemma relates validity in TLA$^+$ and our typed fragment of TLA$^+$ is sound.

Lemma 5.8. Given a ground type γ, we have $\vdash \forall x^{\gamma} : \varphi$ implies $\mathcal{R}(\forall x^{\gamma} : \varphi)$.

Definition 5.9. A *typing hypothesis* $\mathcal{H}(x)$ for a variable x is a premise of the form $x \in e$ or $x = e$, for any expression e where x is not free in e.

The meaning of a type hypothesis for an expression e is that it is an *upper bound* to the possible values of e. Therefore, type-correctness invariants are thus natural candidates for typing hypotheses of global variables. Each quantified variable should be bound by a specific set of its possible values. If we know type hypothesis of (global and local) variables, we often decide the type propositions of most expressions in a TLA$^+$ specification. The challenge here is that typing propositions may appear in a given specification in many different, though equivalent, forms. For example, the typing proposition $[\![S : \mathsf{Set\ Int}]\!]$ is equal to $\forall z \in S : z \in \mathit{Int}$, but it may appear, for instance, as the equivalent formula $S \in \mathsf{SUBSET}\ \mathit{Int}$. Then, it is not always possible to easily identify them. Therefore, we have the following restriction

Every type-correctness invariant is in the form $x \in e$ where e is $\mathit{BOOLEAN}, \mathit{Nat}, \mathit{Int}$ or constructs for SUBSET , a set of function, a set of records and a set of tuples.

5.3.4 Type system

A *type system* is defined by a collection of inference rules which are independent from particular type-checking algorithms [Car96]. Given a TLA$^+$ expression e and an expected type τ for that expression, the main purpose of a TLA$^+$ type system is to generate a valid derivation tree of inference rules in order to decide whether or not τ is a type of e. To explain how to construct a derivation, first we need to provide some definitions.

The description of a type system starts with the description of a collection of judgements. A *judgement* is a triple in the form:

$$\Gamma \vdash e : \tau$$

asserts that the TLA$^+$ expression e has type τ in the typing context Γ. The typing context Γ is usually represented as an ordered list of distinct variables and their types, of the form $\varnothing, x_1 : \tau_1, \ldots, x_n : \tau_n$. The symbol $\varnothing$ is used to denote the empty typing environment.

The collection of variables $x_1, \ldots, x_n$ declared in Γ is indicated by $\mathsf{dom}\,(\Gamma)$. In our system, a variable x can be a variable symbol or an operator symbol in TLA$^+$. A symbol $\Gamma, x : \tau$ is used to emphasiszethe variable x in the typing context and implies that x does not appear in the domain of Γ.

The application of a type assignment σ to a typing context is recursively defined by following rules:

$$\sigma\varnothing = \varnothing \qquad \text{and} \qquad \sigma(\Gamma, x : \tau) = (\sigma\Gamma, x : \sigma\tau).$$

A type inference rule is written as a number of *premise* judgements $\Gamma_i \vdash e_i : \tau_i$ and a finite collection of atomic type *constraints* $\mathcal{C}$ [5] above a horizontal line, with a single *conclusion* judgement $\Gamma \vdash e : \tau$ below the line, where e_i are typically sub-expressions of e. In other words, an inference rule has the form

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma_n \vdash e_n : \tau_n \qquad \mathcal{C}_1 \quad \ldots \quad \mathcal{C}_m}{\Gamma \vdash e : \tau}\, R$$

where R is a rule's name. If all of the premises and constraints are valid, the conclusion must be valid. Therefore, in order to prove that an expression e has a type τ, we need to find a derivation tree with a conclusion $\Gamma \vdash e : \tau$. A symbol e^τ denotes that an expression e has a type τ and a judgement $\Gamma \vdash e : \tau$ is proven.

The definition of the type inference rules for the systems $\mathcal{T}$ is inspired from standard type systems which are usually studied by the type theoretical research community [Car96, Mil78]. During a type derivation, the type inference rules for those systems introduce constraints with many fresh type variables, which are unified throughout to obtain the most general type. In order to find derivation trees and to sovle constraints, our type system needs two main modules which are

- the *unification* module which looks for type assignments of type variables, and
- the *constraint* module which checks whether or not all type constraints are satisfied.

If the type unification module cannot find a type assignment σ for a given expression e, an error message will be showed. If so, then type variables must be instantiated by σ. After that, the constraint module will generate constraints for e and check whether or not σ satisfies all the constraints. If all requirements are satisfied, we can ensure that no errors occur during the next translation step.

A pair $\langle \Gamma, \tau \rangle$ is a typing of an expression e if and only if $FV\,(e) \subseteq dom\,(\Gamma)$ and the prejudgement $\Gamma \vdash e : \tau$ is valid. An expression e is *typable* if and only if it admits a type.

[5]An atomic constraint is defined in the following paragraph "Type inference rule".

Type inference rules

The typing inference rules for Boolean TLA$^+$ expressions are given in Fig 5.2. The types for variables are taken directly from the typing context Γ through rule $\mathcal{T}$-Var. [6].

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ \mathcal{T}\text{-Var} \qquad \frac{}{\Gamma \vdash \mathsf{TRUE} : \mathsf{Bool}}\ \mathcal{T}\text{-True}$$

$$\frac{}{\Gamma \vdash \mathsf{FALSE} : \mathsf{Bool}}\ \mathcal{T}\text{-False} \qquad \frac{\Gamma \vdash \varphi : \mathsf{Bool}}{\Gamma \vdash \neg\varphi : \mathsf{Bool}}\ \mathcal{T}\text{-Not}$$

$$\frac{\Gamma \vdash \varphi_1 : \mathsf{Bool} \qquad \Gamma \vdash \varphi_2 : \mathsf{Bool}}{\Gamma \vdash \varphi_1 \wedge \varphi_2 : \mathsf{Bool}}\ \mathcal{T}\text{-And} \qquad \frac{\Gamma \vdash \varphi_1 : \mathsf{Bool} \qquad \Gamma \vdash \varphi_2 : \mathsf{Bool}}{\Gamma \vdash \varphi_1 \vee \varphi_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Or}$$

$$\frac{\Gamma \vdash \varphi_1 : \mathsf{Bool} \qquad \Gamma \vdash \varphi_2 : \mathsf{Bool}}{\Gamma \vdash \varphi_1 \Rightarrow \varphi_2 : \mathsf{Bool}}\ \mathcal{T}\text{-If} \qquad \frac{\Gamma \vdash \varphi_1 : \mathsf{Bool} \qquad \Gamma \vdash \varphi_2 : \mathsf{Bool}}{\Gamma \vdash \varphi_1 \Leftrightarrow \varphi_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Iff}$$

$$\frac{\Gamma \vdash e : \mathsf{Set}\ \tau \qquad \Gamma, x : \tau \vdash \varphi : \mathsf{Bool} \qquad x \notin FV(e)}{\Gamma \vdash \forall x \in e \,.\, \varphi : \mathsf{Bool}}\ \mathcal{T}\text{-ForAll}$$

$$\frac{\Gamma \vdash e : \mathsf{Set}\ \tau \qquad \Gamma, x : \tau \vdash \varphi : \mathsf{Bool} \qquad x \notin FV(e)}{\Gamma \vdash \exists x \in e \,.\, \varphi : \mathsf{Bool}}\ \mathcal{T}\text{-Exists}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash e_1 = e_2 : \tau_1}\ \mathcal{T}\text{-Eq}$$

Figure 5.2: Inference rules for TLA$^+$ Boolean expressions in $\mathcal{T}$

Inference rules for Boolean operators require their arguments to have the ground type Bool. Quantified formulas in the unbounded always have unknown types since there are no rules form them in Fig. 5.2. Therefore, an error message will be showed if a TLA$^+$ given specification has unbounded quantified formulas. Notice that constraints in inference rules only orient a type assignment or a unification problem.

The Appendix 26 shows remaining inference rules for other operators. When evaluating an expression f that is supposed to be a function, as in the case of rules $\mathcal{T}$-App and $\mathcal{T}$-Dom, we cannot evaluate the type of f by imposing a functional type on it. We expect to derive the desired functional type from the context. That is why we need to define special type operators dom and cod in Section 5.3.1. Strings ($\mathcal{T}$-Str), the set of Boolean values ($\mathcal{T}$-Bool), literal integers ($\mathcal{T}$-Num), the set of integers ($\mathcal{T}$-Int) have a constant

[6]This rule is not for a Boolean TLA$^+$ expression but we represent it here because it is basic.

type. Inference rules for (comparison) arithmetic operators require their arguments to have the ground type Int.

It is easy to see that the typing inference rules in our type systems are syntax-directed. In other words, for any expression e, at most one typing rule may be applied. Therefore, a (possible) type assignment for an expression e is unique and fully determined by the shape of the expression, in a given typing context. In other words, every expression are assigned at most one expected ground type. This property can be easily proved by induction on the structure of e (based on its operator).

Type constraints

A type derivation for a TLA$^+$ expression through inference rules finds a type assignment which marks the TLA$^+$ expression and each of its sub-expressions with a type. In order to construct a tree derivation, all expected types are forced to match type constraints which lies in two equivalence relations on types: unifying $\equiv$ and non-unifying $\cong$ whose definitions are described in the section 5.3.2. An atomic type constraint $\mathcal{C}_A$ is defined by the grammar:

$$\mathcal{C}_A ::= \Gamma \vdash \tau \equiv \tau \mid \Gamma \vdash \tau \cong \tau$$

A constraint $\Gamma \vdash \tau_1 \cong \tau_2$ is *satisfiable* if and only if τ_1 and τ_2 are ground types and have the same syntactical structre. Informally, they are the same object. A constraint $\Gamma \vdash \tau_1 \equiv \tau_2$ is satisfiable, noted $\sigma \vDash \Gamma \vdash \tau_1 \equiv \tau_2$, if and only if there exists a ground assignment σ that makes $\sigma\Gamma \vdash \sigma\tau_1 \equiv \sigma\tau_2$ valid.

From the above definitions and the fact that types are disjoint, it is trivial to derive the following equivalent, purely syntactic rules,

$$\frac{\beta \in \mathcal{S}}{\sigma \vDash \Gamma \vdash \beta \equiv \beta}\ \mathcal{C}\text{-}\equiv\text{-}\beta \qquad \frac{\beta \in \mathcal{S}}{\Gamma \vdash \beta \cong \beta}\ \mathcal{C}\text{-}\cong\text{-}\beta$$

$$\frac{\sigma \vDash \Gamma \vdash \tau_1 \equiv \tau_2}{\sigma \vDash \Gamma \vdash \mathsf{Set}\ \tau_1 \equiv \mathsf{Set}\ \tau_2}\ \mathcal{C}\text{-}\equiv\text{-}\mathsf{SET} \qquad \frac{\tau_1 \cong \tau_2}{\Gamma \vdash \mathsf{Set}\ \tau_1 \cong \mathsf{Set}\ \tau_2}\ \mathcal{C}\text{-}\cong\text{-}\mathsf{SET}$$

$$\frac{\sigma \vDash \Gamma \vdash \tau_1 \equiv \tau_1' \qquad \sigma \vDash \Gamma \vdash \tau_2 \equiv \tau_2'}{\sigma \vDash \Gamma \vdash \tau_1 \to \tau_2 \equiv \tau_1' \to \tau_2'}\ \mathcal{C}\text{-}\equiv\text{-}\mathsf{ARROW}$$

$$\frac{\tau_1 \cong \tau_1' \qquad \tau_2 \cong \tau_2'}{\Gamma \vdash \tau_1 \to \tau_2 \cong \tau_1' \to \tau_2'}\ \mathcal{C}\text{-}\cong\text{-}\mathsf{ARROW}$$

where $\mathcal{S}$ is a set of ground types.

A general constraint is considered as a logical formula and constructed by the following grammar:

$$\mathcal{C} ::= \mathcal{C}_A \mid \mathcal{C} \wedge \mathcal{C} \mid \exists \alpha \,.\, \mathcal{C}$$

In addition to atomic constraints $\mathcal{C}_A$, a general constraint $\mathcal{C}$ is either a conjunction of constraints, or the existential quantification of type variables. The two new constraints allow to reflect the tree structure of a type derivation in a single constraint expression.

Non-atomic constraints are interpreted by the following rules:

$$\frac{\sigma \vDash \mathcal{C}_1 \qquad \sigma \vDash \mathcal{C}_2}{\sigma \vDash \mathcal{C}_1 \wedge \mathcal{C}_2}\ \mathcal{C}\text{-}\wedge \qquad\qquad \frac{\sigma, \alpha \mapsto \gamma \vDash \mathcal{C}}{\sigma \vDash \exists \alpha \,.\, \mathcal{C}}\ \mathcal{C}\text{-}\exists$$

where the mapping $\sigma, \alpha \mapsto \gamma$ updates σ with a new assignment where $\alpha \notin \mathsf{dom}\,(\sigma)$ and γ is a ground type. A constraint $\mathcal{C}$ is *satisfiable*, noted $\sigma \vDash \mathcal{C}$, if and only if there exists a ground assignment σ that satisfies $\mathcal{C}$.

Just for presentational purposes, sometimes we write $\exists \alpha_a \,.\, (\ldots (\exists \alpha_b \,.\, \mathcal{C}))$ as $\exists \alpha_a, \ldots, \alpha_b \,.\, \mathcal{C}$, and a concatenation of constraint conjunctions $\mathcal{C}_1 \wedge (\mathcal{C}_2 \wedge (\ldots \wedge \mathcal{C}_n))$ as a multi-line list of constraints, as in TLA$^+$.

In addition to constraints for types, constraints for expressions' values are used. For example, the rule $\mathcal{T}$-Rcd requires that there are no duplicate values in $h_1, \ldots, h_n$ and this requirement is represented as a constraint of the cardinality of a set $\{h_1, \ldots, h_n\}$, that is, $\{|h_1, \ldots, h_n\}| = n$.

The lack of sub-types is an important feature of the type system $\mathcal{T}$ since it makes $\mathcal{T}$ simple and type constraint problems here can be solved efficiently by a basic unification algorithm [Rob65]. Moreover, since a type assignment found by the unification module assigns every (sub-)expression a ground type, constraints generated by the constraint module are based only the non-unifiable relation $\cong$.

We define a new operator $\langle\!\langle\rangle\!\rangle$, call *constraint generator* (CG), whose main purpose is to generate constraints for given a typing context Γ, a TLA$^+$ expression e, and an expected type τ, with $FV(e) \subseteq dom\,(\Gamma)$. The symbol $\langle\!\langle \Gamma \vdash e : \tau$ denotes generated constraints. The operator is recursively defined over the structure of e. The full definition of $\langle\!\langle\rangle\!\rangle$ is described in the Appendix 26. Here, we explain how CG works through an example with the inference rule $\mathcal{T}$-Mem. The corresponding constraint for $\mathcal{T}$-Mem is defined as the following:

$$\begin{aligned}\langle\!\langle \Gamma \vdash e_1 \in e_2 : \tau \rangle\!\rangle \triangleq \exists \alpha \,.\, & \wedge \langle\!\langle \Gamma \vdash e_1 : \alpha \rangle\!\rangle \\ & \wedge \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\ \alpha \rangle\!\rangle \\ & \wedge \tau \cong \mathsf{Bool}\end{aligned}$$

The constraint is the conjunction of constraints for $\mathcal{T}$-Mem's premises and an extra atomic constraint for the expected type which must Bool. Informally, every free type that appears in the typing rule premises is replaced by a fresh type variable and existentially bound.

The following lemma asserts soundness and completeness of the CG rules with respect to the type inference rules, when the judgements are ground by a type assignment σ.

Lemma 5.10. (CG soundness and completeness). Assuming $FV\,(e) = dom\,(\Gamma)$, then $\sigma \models \langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle$ if and only if $\sigma\Gamma \vdash e : \sigma\tau$, for some ground assignment σ such that every sub-expression of e is assigned a ground type by σ.

In other words, if the constraint $\langle\!\langle \Gamma \vdash e : \tau \rangle\!\rangle$ is satisfied by σ, then the generated constraint contains only valid equality relationships, and the expression e is typable. This lemma can be proved by an induction on the structure of e [MV14].

Type-checking procedure

The Algo. 5.1 shows how our type checker works. First, bounded variables should be renamed such that each has a unique name. Bounded variables in the TLA$^+$ language are local and therefore, one name can be shared between many bounded variables. Renaming is mandatory to avoid conflicts. Second, the type invariant *TypeOK* with the empty environment Γ and the empty assignment σ is passed to `findGroundAssignment` and `checkGroundAssignment` to construct ground type information in $\mathcal{T}$ for remaining variables. Finally, two predicates *Init* and *Next* are checked. Notice that the type-checking procedure for *Init* and *Next* cannot start with the empty environment and the empty assignment. Variables declared by the statement VARIABLES are global and hence, all occurrences of a global one should be assigned the same type.

Derivation examples

We here show three toy examples of type derivation in Examples 5.11, 5.12 and 5.13.

Example 5.11. Let's consider the following formula

$$\{\} \in \{\} \cup \{\{\}, \{1,2,3\}\}$$

Type derivation of the above expression is shown in following proof trees. Because of lack of space, the derivation is divided into sub-steps [1], [2], etc.

$$\dfrac{\dfrac{}{\varnothing \vdash \{\} : \mathsf{Set}\ \tau_1}\ \mathcal{T}\text{-Empty} \qquad [1] \qquad \mathsf{Set}\ \tau_1 \equiv \tau_2}{\varnothing \vdash \{\} \in \{\} \cup \{\{\}, \{1,2,3\}\} : \mathsf{Bool}}\ \mathcal{T}\text{-Mem}$$

Type derivation of [1]:

Algorithm 5.1: Type checking for a TLA$^+$ specification

```
input  : A TLA$^+$ specifiction $Spec$ with the type invariant $TypeOK$
output : A type environment $\Gamma$ and a ground type assignment $\sigma$

Rename bounded variables;
$\Gamma \coloneqq \varnothing, \sigma \coloneqq \square$;

if not findGroundAssignment($TypeOK, \Gamma, \sigma$) then
    showError($TypeOK, \Gamma, \sigma$);
end
if not checkGroundAssignment($TypeOK, \Gamma, \sigma$) then
    showError($TypeOK, \Gamma, \sigma$);
end

if not findGroundAssignment($Init, \Gamma, \sigma$) then
    showError($TypeOK, \Gamma, \sigma$);
end
if not checkGroundAssignment($Init, \Gamma, \sigma$) then
    showError($TypeOK, \Gamma, \sigma$);
end

if not findGroundAssignment($Next, \Gamma, \sigma$) then
    showError($TypeOK, \Gamma, \sigma$);
end
if not checkGroundAssignment($Next, \Gamma, \sigma$) then
    showError($TypeOK, \Gamma, \sigma$);
end
```

$$\dfrac{\dfrac{}{\varnothing \vdash \{\} : \mathsf{Set}\ \tau_2}\ \mathcal{T}\text{-Empty} \qquad [2] \qquad \tau_2 \equiv \mathsf{Set}\ \tau_3}{\varnothing \vdash \{\} \cup \{\{\}, \{1,2,3\}\} : \mathsf{Set}\ \tau_2}\ \mathcal{T}\text{-Cup}$$

Type derivation of [2]:

$$\dfrac{\dfrac{}{\varnothing \vdash \{\} : \mathsf{Set}\ \tau_3}\ \mathcal{T}\text{-Empty} \qquad [3] \qquad \mathsf{Set}\ \tau_3 \equiv \mathsf{Set}\ \mathsf{Int}}{\varnothing \vdash \{\{\}, \{1,2,3\}\} : \mathsf{Set}\ \mathsf{Set}\ \tau_3}\ \mathcal{T}\text{-Enum}$$

Type derivation of [3]:

$$\dfrac{\dfrac{i \in \{1,2,3\}}{\varnothing \vdash i : \mathsf{Int}}\ \mathcal{T}\text{-Num} \qquad \mathsf{Int} \equiv \mathsf{Int} \equiv \mathsf{Int}}{\varnothing \vdash \{1,2,3\} : \mathsf{Set}\ \mathsf{Int}}\ \mathcal{T}\text{-Enum}$$

When the derivation is finished, we gather the list of generated constraints

$$\mathsf{Int} \equiv \mathsf{Int} \qquad \mathsf{Set}\ \tau_3 \equiv \mathsf{Set}\ \mathsf{Int} \qquad \tau_2 \equiv \mathsf{Set}\ \tau_3 \qquad \mathsf{Set}\ \tau_1 \equiv \tau_2$$

After resolving the unifying equalities, we obtain the following assignment σ:

$$\sigma = \{\tau_3 \mapsto \mathsf{Int}, \tau_2 \mapsto \mathsf{Set\ Int}, \tau_1 \mapsto \mathsf{Int}\}$$

Since σ satisfies all constraints and maps every type variable to a ground type, the whole derivation is valid.

Example 5.12. Let's consider the following formula

$$\{\text{“a”}\} \setminus \{\text{“a”}\} = \{1\} \setminus \{1\}$$

Type derivation of the above expression is shown in following proof trees. Because of lack of space, the derivation is divided into sub-steps [1], [2], etc.

$$\dfrac{[1] \qquad [2] \qquad \mathsf{Set}\ \tau_1 \equiv \mathsf{Set}\ \tau_3}{\varnothing \vdash \{\text{“a”}\} \setminus \{\text{“a”}\} = \{1\} \setminus \{1\} : \mathsf{Bool}}\ \mathcal{T}\text{-Eq}$$

Type derivation of [1]:

$$\dfrac{\dfrac{\dfrac{\mathsf{Str} \equiv \tau_1}{\varnothing \vdash \text{“a”} : \tau_1}\ \mathcal{T}\text{-Str}}{\varnothing \vdash \{\text{“a”}\} : \mathsf{Set}\ \tau_1}\ \mathcal{T}\text{-Enum} \qquad \dfrac{\dfrac{\mathsf{Str} \equiv \tau_2}{\varnothing \vdash \text{“a”} : \tau_2}\ \mathcal{T}\text{-Str}}{\varnothing \vdash \{\text{“a”}\} : \mathsf{Set}\ \tau_2}\ \mathcal{T}\text{-Enum} \qquad \tau_1 \equiv \tau_2}{\varnothing \vdash \{\text{“a”}\} \setminus \{\text{“a”}\} : \mathsf{Set}\ \tau_1}\ \mathcal{T}\text{-SetMinus}$$

Type derivation of [2]:

$$\dfrac{\dfrac{\dfrac{\mathsf{Int} \equiv \tau_3}{\varnothing \vdash 1 : \tau_3}\ \mathcal{T}\text{-Int}}{\varnothing \vdash \{1\} : \mathsf{Set}\ \tau_3}\ \mathcal{T}\text{-Enum} \qquad \dfrac{\dfrac{\mathsf{Int} \equiv \tau_4}{\varnothing \vdash 1 : \tau_4}\ \mathcal{T}\text{-Int}}{\varnothing \vdash \{1\} : \mathsf{Set}\ \tau_4}\ \mathcal{T}\text{-Enum} \qquad \tau_3 \equiv \tau_4}{\varnothing \vdash \{1\} \setminus \{1\} : \mathsf{Set}\ \tau_3}\ \mathcal{T}\text{-SetMinus}$$

When the derivation is finished, we gather the list of generated constraints

$$\begin{array}{ccc} \mathsf{Str} \equiv \tau_1 & \mathsf{Str} \equiv \tau_2 & \tau_1 \equiv \tau_2 \\ \mathsf{Int} \equiv \tau_3 & \mathsf{Int} \equiv \tau_4 & \tau_3 \equiv \tau_4 \\ & \mathsf{Set}\ \tau_1 \equiv \mathsf{Set}\ \tau_3 & \end{array}$$

No unifier satisfies all of these above constraints and therefore, the above expression is not allowed in our system.

Example 5.13. Let's consider the following formula

$$\{\} = \{\}$$

Type derivation of [1]:

$$\dfrac{\dfrac{}{\emptyset \vdash \{\} : \mathsf{Set}\ \tau_1}\ \mathcal{T}\text{-Empty} \qquad \dfrac{}{\emptyset \vdash \{\} : \mathsf{Set}\ \tau_2}\ \mathcal{T}\text{-Empty} \qquad \mathsf{Set}\ \tau_1 \equiv \mathsf{Set}\ \tau_2}{\emptyset \vdash \{\} = \{\} : \mathsf{Set}\ \tau_1}\ \mathcal{T}\text{-Eq}$$

When the derivation is finished, we gather one generated constraint $\mathsf{Set}\ \tau_1 \equiv \mathsf{Set}\ \tau_2$ whose most general unifiers are $\sigma_1 = \{\tau_1 \mapsto \tau_2\}$ and $\sigma_2 = \{\tau_2 \mapsto \tau_1\}$. Since no most general unifier maps each type variable to a ground type, the above expression is not allowed in our system.

5.4 Boolification

In TLA$^+$, there is no syntactic difference between Boolean and non-Boolean expressions. Therefore, we need to classify those elements of $\mathcal{V}$ (the set of variable symbols) and $\mathcal{O}$ (the set of operator symbols) that are used as logical formulas and those that are not. In our interpretation of TLA$^+$ expressions, Boolean expressions always have a truth (Boolean) value, and the arguments of logical operators always have a truth value as well.

Example 5.14. While the operator TRUE in the TLA$^+$ expression $p \wedge \mathsf{TRUE}$ should be considered as a Boolean constant, in the TLA$^+$ exprssion $\mathsf{TRUE} \in \{\mathsf{FALSE}\}$, it should not be thought as an operator, but a constant.

Example 5.15. Consider the TLA$^+$ expression $\forall x : (\neg\neg x) = x$, which is not a theorem and whose validity could be easily misinterpreted. The main reason is that it is not possible to decide the truth value of this formula because x could have any value, such as 1 or "a".

Merz and Vanzetto [Van14] introduced an algorithm to identify the symbols used as propositions, which is mutually defined by the operator $[\![e]\!]^+$ that treats the expression e as a formula, and by the operator $[\![e]\!]^-$ that considers e as a non-Boolean expression. The algorithm recursively traverses an expression searching for the arguments of every sub-expression. When it finds an expression e that is implicitly used as a Boolean, it puts a superscript mark ^{b} on e. This only applies if e is a term, a function application, or a CHOOSE expression. In particular, equality yields a Boolean value but it is not expected that its arguments are formulas. If a non-Boolean expression, like a set or an operator, is tried to be Boolified, meaning that a Boolean formula is expected in its place, the algorithm aborts with a "type" error. Finally, an expression e has a Boolean value if and only if it has a ^{b} mark or if it is a formula, that is, an expression of the form $e_1 \circ e_2$, $\forall x : \phi$, $e_1 = e_2$, $e_1 \in e_2$, or an expression defined from formulas where $\circ$ is a logical connective.

The rules of boolification are presented below:

$[\![x]\!]^+ \triangleq x^b$

$[\![w(e)]\!]^+ \triangleq w^b([\![e]\!]^-)$

$[\![e1 \Rightarrow e_2]\!]^+ \triangleq [\![e_1]\!]^+ \Rightarrow [\![e2]\!]^+$

$[\![\forall x : e]\!]^+ \triangleq \forall x : [\![e]\!]^+$

$[\![e_1 = e_2]\!]^+ \triangleq [\![e1]\!]^- = [\![e_2]\!]^-$

$[\![e_1 \in e_2]]+ \triangleq [\![e_1]\!]^- \in [\![e_2]\!]^-$

$[\![e_1[e_2]]\!]^+ \triangleq ([\![e_1]\!]^-[[\![e_2]\!]^-])b$

$[\![E]\!]^+ \triangleq error$

$$[\![\mathsf{IF}\ e_1\ \mathsf{THEN}\ e_2\ \mathsf{ELSE}\ e_3]\!]^+ \triangleq \mathsf{IF}\ [\![e_1]\!]^+\ \mathsf{THEN}\ [\![e_2]\!]^+\ \mathsf{ELSE}\ [\![e_3]\!]^+$$
$$[\![\mathsf{CHOOSE} x : e]\!]^+ \triangleq (choosex : [\![e]\!]^+)^b$$

$[\![x]\!]^- \triangleq x$

$[\![w(e)]\!]^- \triangleq w([\![e]\!]^-)$

$[\![e_1 \Rightarrow e_2]\!]^- \triangleq [\![e_1 \Rightarrow e_2]\!]^+$

$[\![\forall x : e]\!]^- \triangleq [\![\forall x : e]\!]^+$

$[\![e_1 = e_2]\!]^- \triangleq [\![e_1 = e_2]\!]^+$

$[\![e_1 \in e_2]\!]^- \triangleq [\![e_1 \in e_2]\!]^+$

$[\![e_1[e_2]]\!]^- \triangleq [\![e_1]\!]^-[[\![e_2]\!]^-]$

$[\![\mathsf{DOMAIN}\ e]\!]^- \triangleq \mathsf{DOMAIN}\ [\![e]\!]^-$

$[\![[x \in e_1 \mapsto e_2]]\!]^- \triangleq [x \in [\![e_1]\!]^- \mapsto [\![e_2]\!]^-]$

$[\![\{e_1, \cdots, e_n\}]\!]^- \triangleq \{[\![e1]\!]^-, \cdots, [\![e_n]\!]^-\}$

$[\![\{x \in e_1 : e_2\}]\!]^- \triangleq \{x \in [\![e_1]\!]^- : [\![e_2]\!]^+\}$

$[\![\mathsf{UNION}\ e]\!]^- \triangleq \mathsf{UNION}[\![e]\!]^-$

$[\![\{e_1 : x \in e_2\}]\!]^- \triangleq \{[\![e_1]\!]^- : x \in [\![e_2]\!]^-\}$

$[\![\mathsf{SUBSET}\ e]\!]^- \triangleq \mathsf{SUBSET}[\![e]\!]^-$

$$[\![\mathsf{IF}\ e_1\ \mathsf{THEN}\ e_2\ \mathsf{ELSE}\ e_3]\!]^- \triangleq \mathsf{IF}\ [\![e_1]\!]^-\ \mathsf{THEN}\ [\![e_2]\!]^-\ \mathsf{ELSE}\ [\![e_3]\!]^-$$
$$[\![\mathsf{CHOOSE}\ x : e]\!]^- \triangleq (\mathsf{CHOOSE}\ x : [\![e]\!]^+)$$

e extend the above for set and arithmetic operators.

$[\![S_1 \circ S_2]\!]^+ \triangleq [\![S_1]\!]^+ \circ [\![S_2]\!]^+$

$[\![S_1 \star S_2]\!]^+ \triangleq [\![S_1]\!]^- \star [\![S_2]\!]^-$

$[\![S_1 * S_2]\!]^+ \triangleq [\![S_1]\!]^+ * [\![S_2]\!]^+$

$[\![S_1 \circ S_2]\!]^- \triangleq [\![S_1]\!]^- \circ [\![S_2]\!]^-$

$[\![S_1 \star S_2]\!]^- \triangleq [\![S_1]\!]^- \star [\![S_2]\!]^-$

$[\![S_1 * S_2]\!]^- \triangleq [\![S_1 * S_2]\!]^+$

where $\circ$ is an set operator, $\star$ is an arithmetic operator, and $*$ is a Boolean operator.

5.5 Transformation to SMT

First-order formulas with equality have a direct counterpart in SMT-LIB. Moreover, arithmetic operators are built-in in SMT-LIB. Therefore, it suffices to apply a shallow embedding to map TLA$^+$ quantifiers, logical connectors and the equality symbol to their corresponding entities in the language of Z3. However, not each TLA$^{+\mathcal{T}}$ expression has a native counterpart. Specifically, the TLA$^+$ constructs containing second-order sub-expressions, such as $\{x \in S : p\}$ or $[x \in S \mapsto e]$, cannot be directly mapped to first-order sentences. The goal of our translation is to encode TLA$^+$ expressions in the fragment TLA$^{+\mathcal{T}}$ using essentially first-order logic and uninterpreted functions. Our translation ensures that the original and generated formulas are equi-satisfiable.

Except two sorts Bool and Int, other sorts in our type system $\mathcal{T}$ are newly declared sorts. TLA$^+$ functions and operators are declared as uninterpreted functions with sorted arguments. The empty set, function application and set operator $\in$ are three special cases in our translation since they are mapped to many different objects, depending on the context.

We will discuss how to encode and rewrite a TLA$^+$ enxpression in the following subsections. For the full list of rewriting rule, the reader can see Appendix 26.

5.5.1 Set Theory

Since the set theoretical aspect, every expressions in TLA$^+$ are considered as sets. Therefore, our first challenge is how to encode sets efficiently.

The naive way to encode a set in SMT-LIB is to use a characteristic predicate. The following example shows how to encode a set by a characteristic predicates.

Example 5.16. Let A be a set of integers such that $A = \{1, 2\}$. We can encode A by a fresh function

$$(\mathsf{declare-fun\,A\,(Int)\,Bool})$$

and describe what elements are in A by the following assertions

$$\begin{aligned}&\mathsf{A}(1) = \mathsf{TRUE}\\&\mathsf{A}(2) = \mathsf{TRUE}\\&\forall x \in \mathsf{Int}\,.\,(x \neq 1 \wedge x \neq 2) \Rightarrow \mathsf{A}(x) = \mathsf{FALSE}\end{aligned}$$

However, because SMT-LIB does not allow to construct a function whose arguments are functions, we cannot represent sets of sets in this way. Another popular technique to represents a set is to use the array theory. In Z3, an array is declared as $(\mathsf{Array\,Type1\,Type2})$. Hence, a set of elements whose type is A can be encoded as $(\mathsf{Array\,A\,Bool})$ and a set of sets of A elments can be encoded as $(\mathsf{Array\,(Array\,A\,Bool)\,Bool})$ [dMB09]. Now, a set is encoded as a variable in Array , but unfortunately Z3 does not allow to quantify over Array variables. Therefore, in general, we cannot construct an axiom which represents exactly what are in a given set or define a domain of a function.

In [MV12], every set is encoded as an uninterpreted constant, which is a function without arguments. In our work instead of one type U, we use many types. Here, we explain our ideas through Example 5.17.

Example 5.17. Consider two sets of integers $A = \{1, 2\}$, $B = \{3\}$, and a set of sets $S = \{A, B\}$. First, we declare two fresh types SetInt and $\mathsf{SetSetInt}$. Second, we declare three corresponding functions

$$\begin{aligned}&(\mathsf{declare-fun\ A\ ()\ SetInt})\\&(\mathsf{declare-fun\ B\ ()\ SetInt})\\&(\mathsf{declare-fun\ S\ ()\ SetSetInt})\end{aligned}$$

The operator $\in$ is mapped to two functions whose difference is arguments' types

$$\begin{gathered}(\mathsf{declare-fun\ in_{SetInt}\ (Int\ SetInt)\ Bool})\\(\mathsf{declare-fun\ in_{SetSetInt}\ (SetInt\ SetSetInt)\ Bool})\end{gathered}$$

Finally, three assertions are added

$$\begin{aligned}\forall x_1 \in \mathsf{Int}\,.\,\mathsf{in_{SetInt}}(x_1, \mathsf{A}) &\Leftrightarrow (x_1 = 1 \vee x_1 = 2)\\\forall x_2 \in \mathsf{Int}\,.\,\mathsf{in_{SetInt}}(x_2, \mathsf{B}) &\Leftrightarrow (x_2 = 3)\\\forall x_3 \in \mathsf{Set\ Int}\,.\,\mathsf{in_{SetSetInt}}(x_3, \mathsf{S}) &\Leftrightarrow (x_3 = \mathsf{A} \vee x_3 = \mathsf{B})\end{aligned}$$

The main difference between naive approaches and our one is the occurrence of operators "in". In this approach, sets are just constants and their membership relations are formalised via different functions. Therefore, it is possible to represent sets of sets and to quantify over sets.

In our approach, every set type τ has its own operator in_τ. For other set operators such as the intersection $\cap$ or set difference $\setminus$, we can easily define corresponding uninterpreted functions.

While the encoding described above has simple translation rules and is not hard to implement, it has two serious weaknesses. First, the set theory of TLA$^+$ is not finitely axiomatizable. Hence, some TLA$^+$ expressions, such as $\{x \in S : P\}$ and $\{e : x \in S\}$ cannot be encoded as first-order axioms. Second, the above encoding does not perform and scale well in practice; the back-end solvers are unable to prove even simple formula [Van14].

State-of-the-art SMT solvers can solve some quantified formulas by instantiating bounded variables. The result depends heavily on instantiation patterns which are used to manage the generation of ground terms. However, we have not been able to find patterns to attach to the axiom formulas that would significantly improve the performance of SMT solvers.

What we do instead is to perform several transformations to a TLA$^+$ expression to obtain an equisatisfiable formula which is in the basic form and therefore, which could be directly passed to the solvers using the above encoding. We can rewrite a TLA$^+$ expression in many ways. However, it is difficult to know what transformation is better. As a result, we decide to use the following heuristics:

The fewer non-basic expressions and the fewer quantified formulas the translation introduces, the easier for the solvers to find a proof or a counter-model.

First, we decide to represent remaining set operators through the operators in_τ and logical connectives. In other words, rewriting rules which are described in 26 are applied to eliminate complex set operators.

Example 5.18. The expression $x \in S \cap T$ can be rewritten as $x \in S \wedge x \in T$. Hence, we do not introduce a fresh uninterpreted function for the operator intersection $\cap$ since we can encode it by the operator "in" and the logical conjunction.

Second, every set has its own empty set and it is used to reduce the number of quantified formulas.

Example 5.19. Let S_1, S_2 be sets of integers such that $S_1 = \{\}$ and $S_2 = \{\}$. Certainly, we can rewrite these expressions as

$$\begin{gathered}\forall x \in \mathit{Int}\,.\,\neg(x \in S_1))\\ \forall x \in \mathit{Int}\,.\,\neg(x \in S_2))\end{gathered}$$

However, our experiments show that it is better to introduce a new constant "empty"$_{\mathsf{SetInt}}$ which is

$$\begin{gathered}(\mathsf{declare-fun\ empty_{SetInt}}\ ()\ \mathsf{SetInt})\\ \forall x \in \mathsf{Int}\,.\,\neg(\mathsf{in_{SetInt}}(x, \mathsf{empty_{SetInt}}))\end{gathered}$$

and then to assert that S_1, S_2, and "empty"$_{\mathsf{SetInt}}$ are equal.

Finally, other trivial rewriting rules, such as $x \in \{\} \longrightarrow \mathsf{FALSE}$ and $x \cup \{\} \longrightarrow x$, allow to further shorten the expression.

Extensionality axiom Instead of one axiom in the untyped language, here every set type $\mathsf{Set}\ \tau$ has its own extensionality axiom.

$$\forall \mathsf{S}, \mathsf{T} \in \mathsf{Set}\ \tau\,.\ (\forall x \in \tau\,.\,\mathsf{in}_\tau(x, \mathsf{S}) = \mathsf{in}_\tau(x, \mathsf{T})) \Rightarrow \mathsf{S} = \mathsf{T}$$

However, reasoning with extensionality axioms is quite expensive since they force the back-end SMT solvers to generate instances over all values in sorts. To avoid the explosive generation, we instantiate equality expressions $x = y$ whenever possible. In these cases, we say that we *expand* equality. For instance, the following rules are derived from set extensionality and the definition of set constructs.

$$\begin{aligned} S = T \cup U &\xrightarrow{S\,:\,\mathsf{Set}\ \tau} \forall x \in \tau\,.\,x \in S \Leftrightarrow x \in T \vee x \in U\\ S = \{1, 2\} &\xrightarrow{S\,:\,\mathsf{Set\ Int}} \forall x \in \tau\,.\,x \in S \Leftrightarrow x = 1 \vee x = 2\\ S = \{x \in T : P(x)\} &\xrightarrow{S\,:\,\mathsf{Set}\ \tau} \forall x \in \tau\,.\,x \in S \Leftrightarrow x \in T \wedge P(x)\end{aligned}$$

Notice that, as TLC, we work with finite domains. Therefore, we can unfold every set (in theory).

5.5.2 Functions

Our second challenge is how to encode functions. The function construct $[x \in S \mapsto e(x)]$ cannot be mapped directly to a FOL expression. Moreover, first-order functions have no notion of function domain other than the types of their arguments even in sorted languages like MS-FOL.

In principle, TLA$^+$ functions can be encoded in the same way as sets [7]. However, instead of the operator $\in$, we here need to consider the function application and the operator DOMAIN. Again, uninterpreted functions are introduced. Every function type $\tau_1 \to \tau_2$ has its own function application $\mathsf{apply}_{\tau_1 \to \tau_2}$ and operator domain $\mathsf{domain}_{\tau_1 \to \tau_2}$. In other words, the TLA$^+$ function application operator has many instances in a corresponding SMT-LIB specification and its mapping depends on the context which is the types of the function domain and range. Example 5.20 explains informally our idea.

Example 5.20. Consider the following TLA$^+$ expression

$$\begin{aligned} &\wedge f = [x \in \{1,2\} \mapsto 0] \\ &\wedge 1 \in \mathsf{DOMAIN}\, f \\ &\wedge a = f[1] \end{aligned}$$

where a is an integer and f is a function from Int to Int. The second expression ensures that the function application's argument in the third expression is always in the function's domain. To encode the above expressions, we first declare a fresh type $\mathsf{Int\_Int}$. Second, we declare three corresponding functions

$$\begin{aligned} &(\mathsf{declare-fun\ a\ ()\ Int}) \\ &(\mathsf{declare-fun\ f\ ()\ Int\_Int}) \\ &(\mathsf{declare-fun\ apply_{Int\_Int}\ (Int\_Int\ Int)\ Int}) \\ &(\mathsf{declare-fun\ domain_{Int\_Int}\ (Int\_Int)\ Set\_Int}) \end{aligned}$$

Third, following assertions about the domain and function application of f are added

$$\begin{gathered} \mathsf{apply_{Int\_Int}}(\mathsf{f}, 1) = 0 \\ \mathsf{apply_{Int\_Int}}(\mathsf{f}, 2) = 0 \\ \forall x_1 \in \mathsf{Int}\,.\, \mathsf{in_{Int\_Int}}(x_1, \mathsf{domain_{Int\_Int}}(\mathsf{f})) \Leftrightarrow (x_1 = 1 \vee x_1 = 2) \end{gathered}$$

Notice that three above assertions are described the construct of f. Finally, we add

$$\begin{gathered} \mathsf{in_{Int\_Int}}(1, \mathsf{domain_{Int\_Int}}(\mathsf{f})) \\ \mathsf{a} = \mathsf{apply_{Int\_Int}}(\mathsf{f}, 1) \end{gathered}$$

[7] In Z3, we can encode a function as an array. However, this encoding is suitable if the function domain has many elements and we do not need to quantify over functions.

TLA$^+$ functions are total: a function applied to any expression has a value, which is unspecified if the argument is not in the function's domain [Lam02]. For example, the expression $f[0]$ is TLA$^+$ well-formed and has undefined value. Computing the domain of a TLA$^+$ function encoded in a first-order logic is not always easy, leading the provers to failed proof attempts (since the domain can be in form $\{e(x) : x \in S\}$).

Therefore, we expect that the user should state explicitly the type-correctness information, (and the upper bound if possible), of a function and its domain.

The over-approximation on a function can make a TLA$^+$ invalid formula, such as

$$\varphi \triangleq f = [x \in \{1,2\} \mapsto 0] \Rightarrow f[0] < f[0] + 1$$

become a valid one in our typed fragment since $f[0]$ becomes an integer when f is assigned the type $\mathsf{Int} \to \mathsf{Int}$. Fortunately, the untyped fragment TLA$^{+\mathcal{T}}$ restricts that every expression in a given TLA$^+$ specification has a TLC value that implies that the function application's argument is always in the function domain. Therefore, the above formula φ is not allowed. The domain condition can be check automatically with the help of TLC, but it is better to add one formula for the domain condition before using the function application. That is, we should rewrite a formula φ into

$$f = [x \in \{1,2\} \mapsto 0] \Rightarrow (0 \in \mathsf{DOMAIN}\, f \land 0 \in f[0] < f[0] + 1)$$

Hence, we do not need assertions for cases in which the function application's argument is not in the function domain.

The design of an appropriate type system is further complicated by the fact that some formulas, such as $g[x] \cup \emptyset = g[x]$, are actually valid irrespectively of what construct of g is and whether or not the domain condition holds. Unfortunately, we have not found a many-typed system which can handle this formula. Here, our system tries to avoid it. If $g[x]$ is not assigned a ground set type $\mathsf{Set}\ \gamma$, our system will show an error message about type conflict. For the domain condition, the user should check it with TLC manually and add a guard $x \in \mathsf{DOMAIN}\ g$ to his specification. In general, a specification usually needs rewriting many times before the translation starts. The user has to refine types manually, before the actual translation starts.

Functions in TLA$^+$ are those terms f that satisfy a special predicate $IsAFcn(f)$ described in [Lam02].

$$IsAFcn(f) \triangleq f = [x \in \mathsf{DOMAIN}\, f \mapsto f[x]]$$

This predicate in the TLA$^+$ specification characterizes the value of f as being a function. In our typed system, the predicate $IsAFcn$ is true if and only if its argument is assigned a ground type $\tau_1 \to \tau_2, [h_i : S_i]$, or $\langle \tau_i \rangle$. Therefore, we can replace correctly $IsAFcn(e)$ into TRUE or FALSE if we know the type of e.

Like the transformation for sets, we also apply rewriting rules to obtain simpler expressions. For example, we have two following rules

$$[x \in S \mapsto e][a] \longrightarrow e[a]$$
$$[f\ \mathsf{EXCEPT}\ ![x] = y][a] \longrightarrow \mathsf{IF}\ a = x\ \mathsf{THEN}\ y\ \mathsf{ELSE}\ f[a]$$

Function extensionality For each function type $\tau_1 \rightarrow \tau_2$, we add a variant of the function extensionality

$$\begin{aligned}\forall \mathsf{f}, \mathsf{g} \in \tau_1 \rightarrow \tau_2 \,.\, (\ &\land \mathsf{domain}(\mathsf{f}) = \mathsf{domain}(\mathsf{g})\\ &\land \forall x \in \tau_1 \,.\, x \in \mathsf{domain}(\mathsf{f}) \Rightarrow \mathsf{apply}_{\tau_1 \rightarrow \tau_2}(\mathsf{f}, x) = \mathsf{apply}_{\tau_1 \rightarrow \tau_2}(\mathsf{g}, x))\\ &\Rightarrow \mathsf{f} = \mathsf{g}\end{aligned}$$

Functions with many arguments In TLA$^+$, functions can have many arguments, $f[e_1, \cdots, e_n]$. They are those whose domain is a set of tuples and therefore, they can be defined as $f[\langle e_1, \cdots, e_n \rangle]$. Because of implementation reasons, we require the user to use a tuple as an argument, instead of many arguments. Moreover, we don't support $f[y_1 \in S_1, \cdots, y_n \in S_n]$ since we cannot handle with CHOOSE .

5.5.3 Strings

Strings in TLA$^+$ are sequences of characters without any operators except equality defined on them. However, in practice, they are usually used as constants. Therefore, we decide to declare a fresh type Str for strings and to encode each string as a distinct uninterpreted function.

Example 5.21. Suppose that a TLA$^+$ specification contains only three strings "init", "echo" and "accept". In addition to declarations for types and corresponding functions, our encoding adds the following assertion

$$\neg(\mathsf{init} = \mathsf{echo}) \land \neg(\mathsf{init} = \mathsf{accept}) \land \neg(\mathsf{echo} = \mathsf{accept})$$

5.5.4 Tuples and records

TLA$^+$ tuples are also functions whose domain is considered as $1..n$, and it is not possible to distinguish functions and tuples by a syntactic analysis. TLA$^+$ records are also functions whose domain is a set of strings, and who return values of any type. As a consequence, we can encode tuples and records in the same way as functions. However, in our encoding, functions are constrained to have the same type for every element in its codomain. By encoding a tuple or a record as a function, we would discard many ones that have different kind of values in each field. Therefore, we decide to introduce new datatypes for records and tuples and fortunately, Z3 allows us to declare new datatypes easily and to quantify over variables of these types.

Since built-in features of Z3 are used to encode records and tuples, we do not need to add extensionality of axioms for records and tuples. These axioms exists implicitly in the corresponding theories in Z3.

Since records and tuples are used a lot, we avoid many unnecessary axioms. This feature makes reasoning about records and tuples more efficiently.

However, Z3 does not support any features for the domain of a record (or a tuple). Therefore, we need to declare a fresh function $\mathsf{domain}_{\mathcal{T}}$ for every record (and tuple) type.

This encoding has four weaknesses.

- Functions now are treated in a different way from records and tuples.
- The empty sequence cannot be handled since a data-type in Z3 requires at least one field.
- An argument of the record selection should be explicitly a string and an argument of the function application for a tuple should be explicitly an integer.
- Before the mapping, we may need to rename a field's name of a record if necessary. A field of a record in TLA$^+$ is considered as a string can be used as a normal expression. However, a field of a datatype in Z3 is thought as a special sequence of characters and can be used only in the selector function. To avoid ambiguity, fields' name renaming is necessary.

Example 5.22. Two following expressions

$$rcd = [a \mapsto 0, b \mapsto \mathsf{TRUE}] \wedge h \in \mathsf{DOMAIN}\ rcd \wedge x = rcd.h$$
$$rcd = [a \mapsto 0, b \mapsto \mathsf{TRUE}] \wedge h = \text{“a”} \wedge x = rcd.h$$

are well-formed in TLA$^+$. However, an argument of the selector function in Z3 cannot be a variable. Hence, the above expressions cannot be translated into the language of Z3 [8].

Example 5.23. Consider the following TLA$^+$ expression

$$rcd = [a \mapsto 0, b \mapsto \mathsf{TRUE}] \wedge a \in \mathsf{DOMAIN}\ rcd$$

If our encoding introduces a new data type

$$(\mathsf{declare-datatypes}\ ()\ ((\mathsf{Pair}\ (\mathsf{mk-pair}\ (\mathsf{a\ Int})\ (\mathsf{b\ Bool})))))$$

and translate the expression $a \in \mathsf{DOMAIN}\ rcd$ into

$$\mathsf{in}_{\mathsf{Pair}}(\mathsf{a}, \mathsf{domain}_{\mathsf{Pair}}(\mathsf{rcd})),$$

then a conflict at a happens. As a result, we decide to rename fields' name in the data-type declaration and the record selection. At other occurrences, nothing changes. Notice that renaming happens only when the transformation ends.

[8] The technique “definition elimination” described later allows us to replace h by “a” and therefore, we can translate the second expression successfully. However, we still cannot translate the first one.

5.5.5 Conditional statement

The TLA+ expression $\mathsf{IF}\, c\, \mathsf{THEN}\, t\, \mathsf{ELSE}\, u$ can be conveniently mapped by using SMT-LIB conditional operator to $\mathsf{ite(c, t, u)}$. The TLA+ expression CASE can be considered as a nested expression IF THEN ELSE.

$$\begin{aligned} &\mathsf{CASE}\ c_1 \to e_1 \Box \ldots \Box\ c_n \to e_n \Box\ \mathsf{OTHER}\ e_{n+1} \\ \longrightarrow &\mathsf{IF}\ c_1 \mathsf{THEN}\ e_1\ \mathsf{ELSE}\, \mathsf{IF}\ \ldots\ \mathsf{ELSE}\ e_{n+1} \end{aligned}$$

The disadvantage of this approach is that we cannot handle the operator CASE without OTHER .

5.5.6 Definition elimination

An primed variable can appear many times in a TLA+ action and its occurrence usually has the form $x' = \psi$ where ψ is often a non-basic expression. In order to improve the encoding, we introduce an optimization procedure that eliminates definitions, in the sense of the preceding sub-section. The heuristics collects definitions of the form $x = \psi$, and then simply substitute every occurrence of x by ψ in the rest of the context. This substitution is the same as the application of the rewriting rule $x \longrightarrow \psi$. The definitions we want to remove typically occur in the action, that is, the user does not need to introduce them.

Example 5.24. In the TLA+ expression $x' = 1 \wedge y' = x' + 1$, the second appearance of x can be safely replaced by 1. Therefore, the expression can be rewritten as $x' = 1 \wedge y' = 1 + 1$.

This procedure is also used to remove definitions of operators. After applying all substitutions, we must keep definitions of global variables and applied operators. For other terms, we can remove safely their definitions.

Example 5.25. Look at Example 5.24 again. If we remove the definition of x, $x' = 1$, we cannot represent a state successfully.

Example 5.26. Suppose we discard an assumption $\mathsf{DOMAIN}\ f = S$, where f is an element of a set $[S \to T]$. If we want to compare f with another function later, we cannot use the function extensionaliy axiom since we do not know what the domain of f is.

Eliminating definitions, especially with the constructs EXCEPT and DOMAIN , can help us obtain a simpler formula. However, the problem of efficiently eliminating definitions from propositional formulas is a major open question in the field of proof complexity [Avi03]. This procedure can result in an exponential growth in the size of a given formula when applied naively.

In our system, if a variable (or an operator) has many definitions, we always choose the first one. For example, in the following expressions $x = 1 \wedge x = y + 1 \wedge z = x + 1$, the variable x has two definition which are $x = 1$ and $x = y + 1$. Our system just chooses the first one and then replaces $z = x + 1$ by $z = 1 + 1$.

Example 5.27. The technique "definition elimination" rewrites the following TLA$^+$ expression

$$f = [\{1,2\} \mapsto 0] \land f' = [f \text{ EXCEPT } ![0] = 1] \land a' = f'[0]$$

into

$$f = [\{1,2\} \mapsto 0] \land f' = [f \text{ EXCEPT } ![0] = 1] \land a' = [f \text{ EXCEPT } ![0] = 1][0]$$

Apply the rewriting rule for the EXCEPT construct, we have

$$\ldots \land a' = \text{IF } 0 = 0 \text{ THEN } 1 \text{ ELSE } f[0]$$

After replacing $0 = 0$ as TRUE and applying the rewriting rule for IF THEN ELSE , we have

$$\ldots \land a' = 1$$

The resulting expression is simpler than the original one since it does not have any function application.

5.5.7 Miscellaneous

Boolean values

For Boolean values TRUE and FALSE, if their occurrences are boolified as non-logical value, we will try to remove those occurrences by rewriting rules.

Arithmetic expressions

For arithmetic expressions, rewriting rules can help us eliminate some trivial rules such as $x + 0 = x$ or $x < x$ where x is a variable annotated with the type Int.

Constant generation

A specification for a distributed algorithm usually contains a set of objects which are represents as records or tuples. Set constructs in TLA$^+$ give us only general information about elements, but not state explicitly what elements are. Therefore, after the transformation, many sub-formulas of the resulting formula are identical or share the same structure. It causes troubles for the back-end solver to find a proof.

Example 5.28. Consider the following TLA$^+$ expression

$$\begin{aligned}&\land S = \{1,2,3\}\\&\land R = [rnd : \{0,1\}, val : \{\text{ “a”, “b”, “c” }\}]\\&\land f \in [S \rightarrow R]\end{aligned}$$

After a few translation step, we have

$$\begin{aligned}&\ldots\\&\land\ f[1] \in [rnd : \{0,1\}, val : \{\text{ “a”, “b”, “c” }\}]\\&\land\ f[2] \in [rnd : \{0,1\}, val : \{\text{ “a”, “b”, “c” }\}]\\&\land\ f[3] \in [rnd : \{0,1\}, val : \{\text{ “a”, “b”, “c” }\}]\end{aligned}$$

Applying more rewriting rules, we obtain

$$\begin{aligned}&\ldots\\&\land\ f[1].rnd = 0 \lor f[1].rnd = 1\\&\land\ f[1].val = \text{ “a” } \lor f[1].val = \text{ “b” } \lor f[1].val = \text{ “c”}\\&\ldots\end{aligned}$$

In the final expression, sub-expressions related to $f[1], f[2], f[3]$ share the same shape.

Fortunately, if the set does not have so many elements, we can unroll it and declare a uninterpreted function for each element. Therefore, instead of rewriting rules for a set of records, we can use rules for an enumerable set which make the resulting formula shorter and simpler.

Example 5.29. For the set R in Example 5.28, we can replace its definition by $S = \{rcd_1, \ldots, rcd_6\}$ where

$$\begin{aligned}&rcd_1.rnd = 0 \land rcd_1.val = \text{ “a”}\\&rcd_2.rnd = 0 \land rcd_2.val = \text{ “b”}\\&\ldots\\&rcd_6.rnd = 1 \land rcd_2.val = \text{ “c”}\end{aligned}$$

Therefore, the final expression is in the form

$$\begin{aligned}&\ldots\\&\land (f[1] = rcd_1 \lor \ldots \lor f[1] = rcd_6)\\&\land (f[2] = rcd_1 \lor \ldots \lor f[2] = rcd_6)\\&\land (f[3] = rcd_1 \lor \ldots \lor f[3] = rcd_6)\end{aligned}$$

Moreover, constants provide hints for the SMT solver to find a proof. Our experiments show that the appearances of constants can help us save a lot of time.

Syntactical comparison

Obviously, if e_1 and e_2 are syntactically identical, we can rewrite the equality $e_1 = e_2$ as a Boolean constant TRUE. Therefore, for each the equality appearing in the translation, first we try to compare the left-hand side's side with the right-hand side's. If they are the same, the equality is rewritten as TRUE.

Example 5.30. During the translation of the expression $f' = [f \text{ EXCEPT } ![0] = 1]$, thank to the type-correctness information, we can replace the condition $\text{DOMAIN}\, f' = \text{DOMAIN}\, f$ with TRUE.

Since the syntactic tree of a given equality is not too big, the cost of this procedure is not expensive. In practice, it can reduce time the SMT solver needs to solve a problem.

Abstraction

The rewriting process significantly eliminates the number of non-basic operators that occur in an action. However, it is not always possible to obtain a basic formula just by applying rewriting rules since its sub-expressions in the non-basic form do not occur in the same form as the left-hand sides of the rewriting rules. Vanzetto introduces a technique, called abstraction, which can solve the similar problem in a proof obligation automatically [Van14]. Briefly, for every occurrence in a proof obligation φ of a non-basic expression ξ, it introduces in its place a fresh term y, and adds the formula $y = \xi$, giving a definition to y, as an assumption in the appropriate context. The new term y now works as an "abbreviation" for the non-basic expression and the equality woks as its "definition". Hence, we can replace the occurrence of ξ in φ by y. In order to obtain a basic formula, we should systematically apply the abstraction for all non-basic operators in a proof obligation and apply rewriting rules. The resulting formula is equisatisfiable basic formula.

However, the implementation of such a technique is beyound the scope of this thesis.

5.6 Properties of our encoding

The following lemmas show two main properties of our rewriting system. Their proofs are similar with the corresponding theorems in [Van14].

Theorem 5.31. *(TLA^+ , $\longrightarrow$) terminates.*

Theorem 5.32. *(TLA^+ , $\longrightarrow$) is confluent.*

Theorem 5.33. *Every rewriting rule in our rewriting system generates a equisatisfiable formula.*

From the above theorems and the property of relativization, we obtain most important theorems of our translation system. These proofs of these lemma are technically extended from Vanzetto's work [Van14].

Theorem 5.34. *Our type synthesis algorithm is decidable.*

c *Sketch.* Our inference rules are syntactic ones and our algorithm for type synthesis is a the standard unification algorithm.

Theorem 5.35. *The original TLA$^+$ formula and the resulting formula generated by our system are equisatisfiable.*

Sketch. We explain the idea to prove the above lemma.

1. Our type system is an extension of Vanzetto's type system $\mathcal{T}_1$ in [Van14]. Because of the type invariant *TypeOK* in a TLA$^+$ specification, we know possible values which a variable can have. With restrictions in our fragment TLA$^{+\tau}$, we can calculate "calculate" the upper bound of possible values of every expression in a TLA$^+$ specification. Therefore can assign safely type information for every expression, involving the empty set.

2. Our encoding (for sets and functions) and rewriting rules are based on Vanzetto's work [Van14]. Because we assume that an argument of an function application is always in the function domain, we can simplify the way to encode functions and the function application. We automatically add an assertion for this condition in our rewriting rules related to functions. Moreover, we add one rule the construct "set of all" and modify some rules to make the transformation more efficiently. Since Vanzetto's system generates equisatisfiable formulas, our system also keeps equisatisfiability property.

3. For the above reasons, we have that the original TLA$^+$ formula and the resulting formula generated by our system are equisatisfiable.

5.7 Related Work

Researchers have recently made attempts on translating non-temporal part of TLA$^+$ into many-sorted (first-order) logic.

Hansen and Leuschel introduce a framework to translate TLA$^+$ to B for validation with ProB [HL12, LB03]. Later, Plagge and Leuschel integrate the Kodkod high-level interface to SAT-solvers into the kernel of ProB [PL12]. However, ProB is an explicit model checking tool and constructing a predicate abstraction for a B specification is not our focus.

In [MV12], Merz and Vaneztto used first-order logic and uninterpreted functions to encode TLA+ expressions. All Boolean expressions are mapped to the sort Bool. A new sort U (for TLA$^+$ universe) is declared for all non-Boolean expressions, including sets, functions, strings and numbers. This encoding is more versatile than ours since it can handle most aspects of TLA$^+$, even the operator CHOOSE . Sets are just values in the universe of discourse (represented by the sort U in the sorted translation), and it is possible to represent sets of sets and to quantify over sets. For example, consider three sets $A = \{1,2\}$, $B = \{3\}$, and $S = \{A, B\}$. First, the authors declare three constants A^U, B^U, C^U in U and then add three axioms:

$$\forall x^U.\mathsf{in}(x, A^{\mathsf{U}}) \Leftrightarrow (x = \mathsf{int2u}(1) \lor x = \mathsf{int2u}(2))$$
$$\forall x^U.\mathsf{in}(x, B^{\mathsf{U}}) \Leftrightarrow (x = \mathsf{int2u}(3))$$
$$\forall x^U.\mathsf{in}(x, S^{\mathsf{U}}) \Leftrightarrow (x = A^{\mathsf{U}} \lor x = B^{\mathsf{U}})$$

where two operators in and int2u are defined below. Non-logical TLA+ operators are encoded as function or predicate symbols with U-sorted arguments. For example, the operators $\cap$ and $\in$ are encoded in SMT-LIB as the functions $\mathsf{intersection} : \mathsf{U} \times \mathsf{U} \rightarrow \mathsf{U}$ and $\mathsf{in} : \mathsf{U} \times \mathsf{U} \rightarrow \mathsf{Bool}$. The semantics of standard TLA+ operators is defined axiomatically. For example, only the operator $\in$ is primitive in the set theory and expressions with other set operators can be rewritten into appropriate expressions with only the operator $\in$. As a result, the corresponding function in in SMT_LIB will be unspecified and we can express in MS-FOL axioms for other set operators. For instance, the operator intersection has a corresponding axiom $\forall x^{\mathsf{U}}, S^{\mathsf{U}}, T^{\mathsf{U}}.\mathsf{in}(x^{\mathsf{U}}, \mathsf{intersection}(S^{\mathsf{U}}, T^{\mathsf{U}})) = \mathsf{in}(x^{\mathsf{U}}, S^{\mathsf{U}}) \land \mathsf{in}(x^{\mathsf{U}}, T^{\mathsf{U}})$. And the construct for set enumeration $\{e_1, \cdots, e_n\}$, with $n \geq 0$, is an n-ary expression, so we declare separate uninterpreted functions for the arities, like $\mathsf{construct\_3} : \mathsf{U} \times \mathsf{U} \times \mathsf{U} \rightarrow \mathsf{U}$, together with the corresponding axioms. The main weakness of the untyped encoding is that this mechanism introduces many additional quantifiers and defines many “fresh” relations, even for built-in operators in SMT-LIB. These appearances significantly decrease the performance of automated theorem provers.

Fortunately, in some cases, TLA$^+$ formulas can be assigned appropriate types such as Int or Bool. Moreover, types arise informally in any domain to categorize objects according to their usage and behavior, even if we work in an untyped universe [CW85]. For that reason, if we could detect type information or type invariants from the original TLA+ formula, the translation is simpler because we can use directly built-in types and operators of the provers. In [MV14], Merz and Vanzetto propose automated procedures, which are based on type refinements [FP91, XP99], to construct types for TLA+ expressions. This method tries to find type information in a TLA+ specification to encode expressions into both sorted and unsorted languages of automated theorem provers. In [Van14], they extend their refinement procedures with a new dedicated type Map that mimics records and tuples by mapping strings to some other type. This method can reduce the number of quantifiers and to utilize features in SMT solvers. However, this method is undecidable and if their typed system cannot decide an appropriate type for an expression, such as the empty set, they will come back to the untyped encoding. Therefore, their systems are not enough efficient to reason about a next-state predicate with an SMT solver since a proof obligation is usually more “shallow” than a next-state predicate.

5.8 Conclusions

In this chapter, first we have described our typed fragment of TLA$^+$, which is named TLA$^{+\tau}$, and its corresponding type system. We have given examples to explain its restrictions and how to assign type information to a TLA$^+$ expression. While our fragment is smaller than what TLC can evaluate, we believe that it is expressive enough to specify many distributed algorithms.

Second, we have extended the Boolification algorihm which Vanzetto suggested in [Van14]. This extension is complete for our fragment and is used to perform a quick double-check of expressions' types.

Moreover, we have presented how to transform an TLA$^+$ expression in our fragment to the language of SMT solvers. Our encoding can handle functions, conditional statements and sets of sets. We use built-in mechanism of Z3 to capture records and tuples and this encoding helps us avoid to introduce unnecessary axioms. Moreover, we have introduces some heuristics to make our translation more efficiently, such as definition elimination or unfolding finite sets.

Finally, we have compared our work with others. Our system is decidable, keeps equisatisfiability and generates formulas which is simple enough to reason with SMT solvers, such as Z3.

CHAPTER 6

The Implementation

This section describes the prototype's architecture and provides a short user manual.

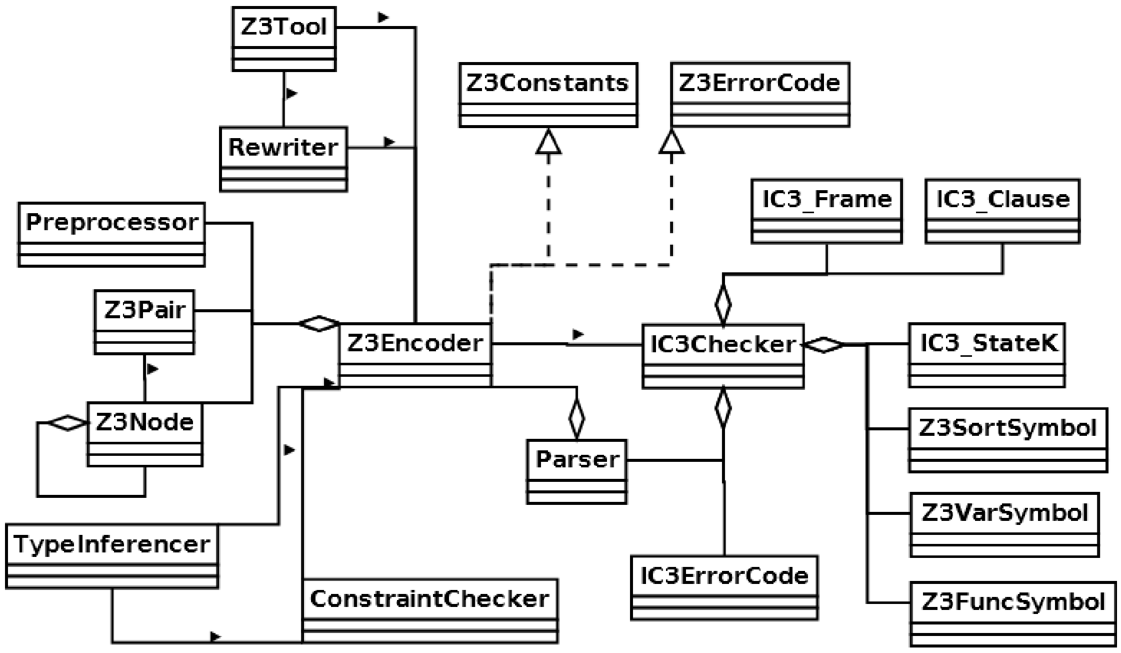
6.1 The Architecture

In our implementation, variables with the prefix `p` or `next` are actions or variables at the next states. Figure 6.1 shows the class diagram of our system

Our system contains the following classes:

1. `Preprocessor` creates new names for TLA$^+$ actions $\mathit{action}\,(v_i)$ into which TLA$^+$ Toolbox splits a "big" action $\exists x \in \{v_1, \ldots, v_n\}\,.\,\mathit{action}\,(x)$.
2. `Z3Constants` contains global constants and operator codes.
3. `Z3ErrorCode` contains codes of errors which may happen during the translation time.
4. `Z3Node` is a "composited" class which stores and summaries information of classes in package SANY such as `OpApplNode, NumeralNode` or `OpDefNode`.
5. `Z3Pair` binds a variable to its new definition which appears in an action.
6. `TypeInferencer` has a list of type inference rules.
7. `ConstraintChecker` has a list of rules for type constraints.
8. `Rewriter` has a list of rewriting rules.

Figure 6.1: Class diagram of our system



9. `Z3Encoder` is one of main classes for translation from TLA$^+$ language to SMT-LIB. It has an object `Tool` which contains all information about modules, variables, invariants and actions which are generated by `SANY`. Based on these materials, `Z3Encoder` builds its semantic trees.

10. `Z3Tool` constructs corresponding types and applies rewriting rules to transform and simplify semantic trees obtained from `Z3Encoder`.

11. `Z3SortSymbol` (or `Z3VarSymbol, Z3FuncSymbol`) is a declaration of a fresh sort (or a variable, a function) in the Java API of Z3.

12. `IC3_StateK`, or $\langle q, k \rangle$ in the algoritm 4.3, is an abstract and bad state `formula` which is found at the `k`-th frame and has a next bad state `next`.

13. `IC3_Clause` is a clause c with one unique `id`. Because of the performance, `formula` may be in the form $\neg(l_1 \wedge \ldots \wedge l_n)$, not $l_1 \vee \ldots \vee l_n$.

14. `IC3_Frame` is a frame F in the algorithm 4.1. Except F_0, other frames are always in CNF. `clauses` contains all clauses in `formula`. While `clauses` can be got back from `formula`, we decide to save both of them to get $c \in F_i$ easy and to reduce the number of the SAT checking of $F_i \wedge T \wedge \neg c'$ in `propagateClauses`.

15. `IC3_ErrorCode` contains codes of errors which may happen during the verification time.

16. `IC3_Worker` is the main part of the algorithm IC3.

6.2 How to use our system

The main features of our system is to test invariants, to find a stronger inductive strengthening and to check safety properties. In order to do that, the user needs to add the following information in the TLA$^+$ specification

- `TypeOK`: contains annotations of every variable's type which should be the minimum set of possible values which a variable can be assigned. For example, if the number of processes is N and p is a process variable, it is better to write $p \in 1 \,..\, N$ than $p \in \mathit{Int}$. Take notice of natural numbers, instead of $x \in \mathit{Nat}$, the user needs to write $x \in \mathit{Int} \land x \geq 0$.
- Predicates $\texttt{pred}_1, \ldots, \texttt{pred}_n$: are used to construct the abstraction. They are declared as user-defined operators.
- `Inv_ToCheck` is a safety property the system needs to check. `Inv_ToCheck` have to use only above predicates $\mathit{pred}_1, \ldots, \mathit{pred}_n$.
- "Fake" invariant `Predicates` is in form $\top \lor \texttt{pred}_1 \lor \ldots \lor \texttt{pred}_n$. It is a hint for our system on what predicates are used to construct the abstraction.

If `Inv_ToCheck` is a safety property, its inductive strengthening is printed. And if `Inv_ToCheck` is an inductive invarient, a constant TRUE is printed. Finally, if a given TLA$^+$ specification violates `Inv_ToCheck`, our system shows a abstract bad path.

6.3 Example

The following specification is the encoding of Chandra's algorithm for reliable broadcast by message diffusion (BcastFolklore) [CT96]. This encoding is based on [KVW15] and this algorithm is used in our experimental evaluation. For the description of this algorithm, we refer the user to Chapter 7.

This encoding introduces many predicates $\mathit{pred}_1, \ldots, \mathit{pred}_21$, two invariants *Inv_ToCheck* and *Predicates*. Moreover, a big action *Step* is manually split into 4 simpler actions *Receive_UponV1*, *Receive_UponCrash*, *Receive_UponAccept* and *Receive_Nothing*.

```
──────────── MODULE bcastFolklore_lazyValues_411 ────────────

EXTENDS Naturals, FiniteSets
```

CONSTANTS $N,\ T,\ F$

we need to say explicitly values of constants.

$N \triangleq 4$

$T \triangleq 1$

$F \triangleq 1$

variable declarations

VARIABLE $pc,\ rcvd,\ sent,\ nfailed$

ASSUME $N \in Nat \land T \in Nat \land F \in Nat$

ASSUME $(N > 2 * T) \land (T \geq F) \land (F \geq 0)$

$P \triangleq 1 \,..\, N$ all processess, including the faulty ones

$Corr \triangleq 1 \,..\, N$ correct processes

$M \triangleq \{\text{"ECHO"}\}$ only messages ($ECHO$) is sent in this algorithm

$PM \triangleq (P \times M)$ inform that our prototype should declare elements of these set

$SUBSETPM \triangleq \text{SUBSET } PM$

predicates are used to construct the abstraction

$pred1 \triangleq pc[1] = \text{"V0"}$ $pred2 \triangleq pc[1] = \text{"V1"}$ $pred3 \triangleq pc[1] = \text{"AC"}$

$pred4 \triangleq pc[1] = \text{"CR"}$ $pred5 \triangleq rcvd[1] = \{\}$

$pred6 \triangleq pc[2] = \text{"V0"}$ $pred7 \triangleq pc[2] = \text{"V1"}$ $pred8 \triangleq pc[2] = \text{"AC"}$

$pred9 \triangleq pc[2] = \text{"CR"}$ $pred10 \triangleq rcvd[2] = \{\}$

$pred11 \triangleq pc[3] = \text{"V0"}$ $pred12 \triangleq pc[3] = \text{"V1"}$ $pred13 \triangleq pc[3] = \text{"AC"}$

$pred14 \triangleq pc[3] = \text{"CR"}$ $pred15 \triangleq rcvd[3] = \{\}$

$pred16 \triangleq pc[4] = \text{"V0"}$ $pred17 \triangleq pc[4] = \text{"V1"}$ $pred18 \triangleq pc[4] = \text{"AC"}$

$pred19 \triangleq pc[4] = \text{"CR"}$ $pred20 \triangleq rcvd[4] = \{\}$

$pred21 \triangleq sent = \{\}$

$vars \triangleq \langle pc,\ rcvd,\ sent,\ nfailed \rangle$

$Receive(self) \triangleq$

$\exists\, r \in SUBSETPM :$

$\quad \land\; r \subseteq sent$

$\quad \land\; rcvd[self] \subseteq r$

$\quad \land\; rcvd' = [rcvd \text{ EXCEPT } ![self] = r]$ receive set ”r“ of msgs

$UponV1(self) \triangleq$

$\land\; pc[self] = \text{“V1”}$ if a process ”has received a msg from a bcasting process and has not sent ($ECHO$)“

$\land\; pc' = [pc \text{ EXCEPT } ![self] = \text{“AC”}]$ it accepts and sends ($ECHO$) to all

$\land\; sent' = sent \cup \{\langle self, \text{“ECHO”}\rangle\}$

$\land\; nfailed' = nfailed$ a number of crashed processes does not change

$UponCrash(self) \triangleq$

$\land\; nfailed < F$ if a number of crashed processes $< F$ and

$\land\; pc[self] \neq \text{“CR”}$ this process is correct, it will be crashed

$\land\; nfailed' = nfailed + 1$ increase a number of crashed processes

$\land\; pc' = [pc \text{ EXCEPT } ![self] = \text{“CR”}]$ update labels of processes

$\land\; sent' = sent$ message channel does not change

$UponAccept(self) \triangleq$

$\land\; (pc[self] = \text{“V0”} \lor pc[self] = \text{“V1”})$ if a process ”has not receive any msg“ or

$\land\; rcvd'[self] \neq \{\}$ ”has received a msg from a bcasting process and has not sent ($ECHO$)“

$\land\; pc' = [pc \text{ EXCEPT } ![self] = \text{“AC”}]$ it accepts and sends ($ECHO$) to all

$\land\; sent' = sent \cup \{\langle self, \text{“ECHO”}\rangle\}$

$\land\; nfailed' = nfailed$ a number of crashed processes does not change

actions

$Receive\_UponV1 \triangleq \exists\, self \in Corr : (Receive(self) \land UponV1(self))$

$Receive\_UponCrash \triangleq \exists\, self \in Corr : (Receive(self) \land UponCrash(self))$

$Receive\_UponAccept \triangleq \exists\, self \in Corr : (Receive(self) \land UponAccept(self))$

$Receive\_Nothing \triangleq \exists\, self \in Corr : (Receive(self) \land pc' = pc \land$
$sent' = sent \land nfailed' = nfailed)$

$Next \triangleq$

$\lor\ Receive\_UponV1$

$\lor\ Receive\_UponCrash$

$\lor\ Receive\_UponAccept$

$\lor\ Receive\_Nothing$

$Init \triangleq$

$\land\ sent = \{\}$ message channel is empty

$\land\ pc \in [Corr \rightarrow \{\text{"V0"}, \text{"V1"}\}]$ process can be labeled as "has not received any msgs" or "has received a msg from a bcasting process and has not sent (*ECHO*)"

$\land\ rcvd = [i \in Corr \mapsto \{\}]$ every process has not received any msg

$\land\ nfailed = 0$ no process crashes

$InitNoBcast \triangleq$

$\land\ sent = \{\}$ message channel is empty

$\land\ pc \in [Corr \rightarrow \{\text{"V0"}\}]$ no process has received msgs from a bcasting process

$\land\ rcvd = [i \in Corr \mapsto \{\}]$ every process has not received any message

$\land\ nfailed = 0$ no process crashes

$Spec \triangleq Init \land \Box[Next]_{vars}$ we will check this specification

just another specification, also used in our benchmarks

$SpecNoBcast \triangleq InitNoBcast \land \Box[Next]_{vars}$

type invariant

$TypeOK \triangleq$

$\land\ sent \in SUBSETPM$

$\land\ pc \in [Corr \rightarrow \{\text{"V0"}, \text{"V1"}, \text{"AC"}, \text{"CR"}\}]$

$\land\ rcvd \in [Corr \rightarrow SUBSETPM]$

$\land\ nfailed \in Nat$

a safety property: no message is sent or all processes do not accpet

$Inv\_ToCheck \triangleq$

$\lor\ (\neg pred21)$

$\lor\ (\ \land (\neg pred3) \land (\neg pred8) \land (\neg pred13) \land (\neg pred18))$

inform our system that these predicates are used and *Predicates* is marked as an invariant

$Predicates \triangleq$

$\lor\ \text{TRUE}\quad \lor\ pred1\quad \lor\ pred2\quad \lor\ pred3\quad \lor\ pred4$

$\lor\ pred5\quad \lor\ pred6\quad \lor\ pred7\quad \lor\ pred8\quad \lor\ pred9$

$\lor\ pred10\quad \lor\ pred11 \lor pred12 \lor pred13 \lor pred14$

$\lor\ pred15\quad \lor\ pred16 \lor pred17 \lor pred18 \lor pred19$

$\lor\ pred20\quad \lor\ pred21$

system properties

$UnforgLtl \triangleq (\forall\, i \in Corr : pc[i] = \text{“V0”}) \implies \Box(\forall\, i \in Corr : pc[i] = \text{“AC”})$

$CorrLtl \triangleq (\forall\, i \in Corr : pc[i] = \text{“V1”}) \implies \Diamond(\exists\, i \in Corr : pc[i] = \text{“AC”})$

$RelayLtl \triangleq \Box((\exists\, i \in Corr : pc[i] = \text{“AC”}) \implies \Diamond(\forall\, i \in Corr : pc[i] = \text{“AC”}))$

CHAPTER 7

Experimental Evaluation

To test our model checker, we used three benchmarks about a threshold automata TA [KVW15] and Chandra's algorithm for reliable broadcast by message diffusion (BcastFolklore) [CT96] and Raynal's algorithm for non-blocking atomic commitment (NBAC) [Ray97]. The benchmarks were provided by my supervisors. They are are based on the repository: https://github.com/konnov/fault-tolerant-benchmarks [KVW15].

Experimental setup: All of the experiments were performed on an Intel Core i5 4210U processor with 4 GB RAM and Windows 7. We ran experiments with TLC and experiments our prototype (predicate abstraction and IC3). In our experiments, time is measured in minutes and T/O is an abbreviation of timeout. The time limitation depends on an experiment.

7.1 Experiment with TA

Figure 7.1 represents an example of a threshold automaton TA over two shared variables $\Gamma = \{x, y\}$ and parameters $\Pi = \{n, t, f\}$ [1]. In our example, n is a number of processes, t is a maximum number of faulty processes, f is a real number of faulty processes and correct $n - f$ processes concurrently execute the automaton TA. The circles depict the local states $\{l_1, \dots, l_5\}$, two of them are the initial states $\{l_1, l_2\}$. The edges depict the transition rules $\{r_1, \dots, r_6\}$, whose labels have the form $\varphi \mapsto \texttt{act}$, where φ is one of the threshold guards:

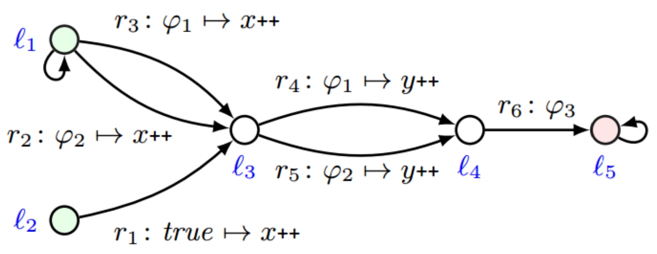
$$\varphi_1 : x \geq \lceil (n+t)/2 \rceil - f, \quad \varphi_2 : y \geq (t+1) - f, \quad \varphi_3 : y \geq (2 * t + 1) - f,$$

and an action act updates the shared variables by increasing x or y, or does nothing (as in rule r_6). Every local state l_i has a non-negative counter $\kappa[i]$ that represents the

[1] This description is from [KVW15].

number of processes in local state l_i. Together with the values of x, y, n, t and f, the values of the counters $\kappa[1], \ldots, \kappa[5]$ constitute a configuration of the system.

Figure 7.1: An example threshold automaton [KVW15]


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

We encoded this transition system in two different ways. In the first approach, we focused on only the number of processes in every local states. That is, processes were not encoded in our first TLA$^+$ specification. The correct specification is named as $Spec_1$. Moreover, we wrote a wrong specification, named as $WSpec_1$ with some bug of counters in transition relations. In the second approach, processes were really encoded. For the second approach, a right specification is named as $Spec_2$ and a wrong one is $WSpec_2$.

We tried to verify two following invariants:

i. inv_1: the number of processes at every states is always $n - f$.

ii. inv_2: assuming that no processes are in local state l_2 in initial steps, we have that $\kappa[5]$ is always 0.

Experiment with $Spec_1$ and inv_1

We tried to check inv_1 with three predicates [2]

$$pred_0 \triangleq \kappa[1] + \kappa[2] + \kappa[3] + \kappa[4] + \kappa[5] = n - f, \quad pred_1 \triangleq x \geq 0, \quad pred_2 \triangleq y \geq 0.$$

Table 7.1 shows running time of our model checker and TLC for $Spec_1$ and inv_1. If n is small ($n < 100$), TLC (and its explicit-state enumeration) works better since our tool spends time translating the TLA$^+$ specification and calling Z3 to verify inv_1. However, if n is large enough ($n \geq 200$), our model checker is more efficient than TLC.

Experiment with $Spec_1$ and inv_2

In this scenario, the explicit-state enumeration is an appropriate choice since a number of reachable state is only 1 for any N. In this scenario, no processes move. We decided to ran experiments with $N = 500$ and 3 predicates

[2]If we use only p_0, wrong conclusion may happen.

Table 7.1: Experiments with $Spec_1, inv_1$, TLC and our model checker

	TLC		Predicates + IC3		
N	time (m)	#states	time (m)	#predicates	#frames
50	1	220701	2	3	3
100	2	318026	2	3	3
150	5	1555851	2	3	3
200	15	48310926	2	3	3
250	42	116712876	2	3	3

$$
\begin{aligned}
&pred_3 \triangleq \kappa[1] = N - F \land \kappa[2] = 0 \land \kappa[3] = 0 \land \kappa[4] = 0 \land \kappa[5] = 0 \land x = 0 \land y = 0,\\
&pred_4 \triangleq x \geq \lceil (n+t)/2 \rceil - f,\\
&pred_5 \triangleq y \geq (t+1) - f,\\
&pred_6 \triangleq y \geq (2 * t + 1) - f
\end{aligned}
$$

Both TLC and our model checker could prove inv_2 in less than 1 minutes. However, in this case TLC is a better choice because we do not think about predicates and TLC runs faster.

Experiment with $WSpec_1$

For inv_1, a concrete error trace is short and easy to find. TLC spent less than 2 minutes to detect a bug, even if $n = 500$. Our model checker need about 2 minutes to find an abstract error trace. However, if we use only a few of predicates, it is not easy to construct a concrete trace from an abstract trace. Therefore, it is difficult to check whether an abstract error trace is spurious or not. Moreover, it took time to find a good set of predicates.

For inv_2, TLC can check very fast whether inv_2 is an invariant or not. Again, about 1 minutes. We did not run experiments with our model checker since we believe that it is better to use TLC in this case.

Experiment with $Spec_2$

When processes are really encoded, TLC needed more time to verify both invariants but our model checker still runs fast. Table 7.2 shows the running time and extra information of TLC and our tool. For the case N = 15, we decided to stop TLC after 24 hours since TLC could not finish its work.

7.2 Experiment with BcastFolklore

Algorithm 7.1 shows the core logic of the Folklore Reliable Broadcast Algorithm for a correct process from [CT96]. We use the benchmark introduced in Ph.D thesis from by

Table 7.2: Experiments with $Spec_2, inv_1$, TLC and our model checker. T/O is 24 hours.

	TLC		Predicates + IC3		
N	time (m)	#states	time (m)	#predicates	#frames
7	1	22680	1	3	3
9	2	408148	1	3	3
11	7	7258975	1	3	3
13	150	127700089	2	3	3
15	T/O	T/O	2	3	3

Annu Gmeiner [Gme15]. A process broadcasts a message by sending it to all processes. Upon the reception of a message for the first time by a process, it sends the message to all the processes in the system and accepts it.

Algorithm 7.1: Core logic Folklore Broadcast Algorithm for correct process i

```
21 $pc_i \in \{V0, V1\}$ ;
22 if ($received\ \langle echo \rangle\ from\ some\ other\ process\ and\ not\ sent\ \langle echo \rangle\ before$) or
23     ($pc_i = V1$) then
24     send $\langle echo \rangle$ to all ;
25     $pc_i = AC$ ;
26 end
```

We use the variable pc_i to show a status of a process. $pc_i = V1$ indicates that the process i has received the message from the broadcasting process and $pc_i = V0$ indicates otherwise. Thus, if a process starts with $pc_i = V1$, and it has not sent an $\langle echo \rangle$ yet, it sends an $\langle echo \rangle$ to every process and accepts. Also, if a process has received $\langle echo \rangle$ and has not sent $\langle echo \rangle$ yet then it sends an $\langle echo \rangle$ to every process and accepts. In this algorithm, there exist some faulty process which is labeled with $pc_i = CR$. The number of faulty processes is less than a half of processes.

For this algorithm, we technically modified a specification "bcastFolklore.tla" and checked

- an invariant bf_inv "Nothing is sent or all processes do not accept.", and
- a property bf_p "All processes do not accept." which is not an invariant.

Table 7.3 shows the running time and extra information of TLC and our tool for an invariant bf_inv. N is a number of processes, T is a maximum number of crashed processes and F is a number of crashed processes. For two cases $(T = 1, F = 1, N = 5)$ and $(T = 1, F = 1, N = 6)$, we decided to stop TLC after 2 hours since TLC could not finish his work. For case $(T = 1, F = 1, N = 7)$, we decided to stop TLC after 1 days. To prove this invariant bf_inv, we use $4 \times N + 1$ predicates. There are 5 predicates for each

Table 7.3: Experiments with *bcastFolklore*, *bf_inv*, TLC and our model checker. T/O is 2 hours and _ is 24 hours.

#processes			TLC		Predicates + IC3		
N	T	F	time (m)	#states	time (m)	#predicates	#frames
2	0	0	1	36	1	11	3
3	1	1	1	1000	1	16	3
4	1	1	3	501552	1	21	3
5	1	1	T/O	?	2	26	3
6	1	1	T/O	?	2	31	3
7	1	1	_	?	2	36	3

process i: $pc_i^1 = V0, pc_i^1 = V1, pc_i^1 = AC, pc_i^1 = CR$ and $rcvd_i = \{\}$. $rcvd_i$ in our encoding is a set of received messages of process i and the last predicate shows whether process i received some message or not. One additional predicate for a message channel $sent = \{\}$.

We use the same set of predicates to check the property bf_p. Since a concrete error trace is not complicated, TLC and our model checker can find a bug in one minutes for the test case $\langle N = 7, T = 1, F = 1\rangle$.

7.3 Experiment with NBAC

In this experiment, Raynal's algorithm for asynchronous non-blocking atomic commitment (NBAC) was considered [Ray97]. Non-blocking atomic commitment (NBAC and NBACC) [29,14]. Here, N processes are initialized with Yes or No and decide on whether to commit a transaction. The transaction must be aborted if at least one process is initialized to No. In this algorithm, a process may crash. Both models contain four shared variables. The algorithm uses a failure detector, which is modeled as local variable that changes its value non-deterministically.

For this algorithm, we technically modified a specification "nbac.tla" and checked an invariant inv_{nbac} "If some process votes NO, no processes commit." The main feature of this algorithm is that a process commits if and only if if a vote YES has been received from all participants, and ABORT in all other cases.

Table 7.4 shows the results with TLC. For case $N = 5$, we stopped TLC after 12 hours since it could not finish his work.

Our model checker worked worse than TLC in this experiment. For case $N = 3$, our checker could not answer whether inv_{nbac} is an invariant or not after 60 minutes. One problem is that process i needs to check whether it has received vote YES from all participants. In our specification, this condition is encoded as

Table 7.4: Experiments with *nbac* and TLC. T/O is 1 hour and _ is 12 hours.

	TLC		Predicates + IC3		
N	time (m)	#states	time (m)	#predicates	#frames
3	1	7352	T/O	?	?
4	10	2406640	T/O	?	?
5	_	?	T/O	?	?

$$\{p \in P : (\exists\, msg \in rcvd'[self] : msg[1] = p \wedge msg[2] = \text{“YES”})\} = P$$

where P is a set of processes, *rcvd* is a message channel and *rcvd*[*self*] is a set of messages which process *self* has received. Unfortunately, a formula generated by our system is too difficult for Z3 to evaluate and our predicates are not good enough to guide Z3.

It is future work to find out, why our tool does not work efficiently for this case study.

7.4 Discussions

Our preliminary experiments show that our new prototype is much more efficiently than TLC, which performs explicit-state model checking.

CHAPTER 8

Conclusion

What we have presented in the preceding chapters is an efficient way of verifying a safety property of a TLA$^+$ specification with predicate abstraction and inductive invariants. The main outcome is a prototypical model checker which can automatically construct an abstract model for the TLA$^+$ specification from a set of predicates given by the user and check the safety property by finding a formal proof for a corresponding inductive invariant.

We have defined a simple type system for a fragment of the TLA$^+$ language. This type system is based on many-sorted first-order logic and can be represented in the SMT-LIB language. Our fragment does not allow the user to use many features in TLA$^+$, such as the CHOOSE operator, recursive functions and a set of elements which are records with different structures. Therefore, our system cannot handle the Paxos specification in [Lam02]. However, we believe that the fragment is enough expressive to describe a wide variety of distributed and concurrency algorithms. An interesting theoretical question is whether we can extend the current type system or utilize the untyped encoding in order to capture more features in TLA$^+$.

We have described rewriting rules to eliminate “complex” TLA$^+$ formulas which do not counterparts in MS-FOL (and SMT-LIB). Many rewriting rules are applied to transform these formulas into simple forms which can be mapped directly to the language of a target SMT solver. During the translation, many heuristics, such as definition elimination, set enumeration and constant declarations, are called to simplify the obtained specification. The translation plays a central role in our work and therefore, every improvement in the translation can bring significant benefits.

We have shown how to use IC3 and predicate abstraction to verify the safety property of the TLA$^+$ specification. For our benchmarks, our model checker outperforms than TLC in finding an inductive variants. The shortcomings of our tool are that the way it shows error traces and that user needs to check whether every abstract state in frame F_k can be reachable in at least k step.

Future work

The translation can be improved in several aspects. First, we are studying how to translate the Cardinality operator for finite sets. Second, if no set of sets occur in the specification, sets can be encoded simply through their characteristic predicates. This encoding can reduce the number of constants and functions in the SMT specification and make the reasoning more simple and efficient. Moreover, there may exist many definitions for a variable in an action and we should find some measure to compare these definitions. Finally, we intend to do more experiments to know the effect of the order of formulas in the SMT solver.

At the moment, our implementation is for a sequence algorithm of IC3 in [Bra11]. Parallel versions of IC3 have been recently introduced in [Bra11, CK16]. These algorithms differ in the degree of synchronization and technique to detect and share lemmas. An important area of future work is to study how effectively these paralleled versions of IC3 can check distributed systems.

Type Inference Rules

This appendix shows type inference rules for sets, arithmetic, functions, records and tuples. Some inference rules for set and operators, such as the intersection operator *cap* or div, are missed but we believe that the reader can construct these rules easily. Missing rules look very similar with rules $\mathcal{T}$-Cup, $\mathcal{T}$-Plus, and $\mathcal{T}$-Less. A constraint $|\{h_1, \ldots, h_n\}| = n$ means that every string $h_1, \ldots, h_n$ is distinct or $h_1 \neq h_n$ for all $i, j \in 1 \mathrel{..} n$ and $i \neq j$.

$$\frac{}{\Gamma \vdash \{\} : \mathsf{Set}\ \alpha}\ \mathcal{T}\text{-Empty}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\ \tau}{\Gamma \vdash \mathsf{SUBSET}\ S : \mathsf{Set}\ \mathsf{Set}\ \tau}\ \mathcal{T}\text{-SUBSET}$$

$$\frac{}{\Gamma \vdash \mathit{Boolean} : \mathsf{Set}\ \mathsf{Bool}}\ \mathcal{T}\text{-Boolean}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\ \mathsf{Set}\ \tau}{\Gamma \vdash \mathsf{UNION}\ S : \mathsf{Set}\ \tau}\ \mathcal{T}\text{-UNION}$$

$$\frac{}{\Gamma \vdash \mathit{Int} : \mathsf{Set}\ \mathsf{Int}}\ \mathcal{T}\text{-Int}$$

$$\frac{n \in \{\ldots, -1, 0, 1, \ldots\}}{\Gamma \vdash n : \mathsf{Int}}\ \mathcal{T}\text{-Num}$$

$$\frac{}{\Gamma \vdash \text{“abc”} : \mathsf{Str}}\ \mathcal{T}\text{-Str}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \qquad \Gamma \vdash \tau_1 \equiv \ldots \equiv \tau_n}{\Gamma \vdash \{e_1, \ldots, e_n\} : \mathsf{Set}\ \omega\,(\tau_1, \ldots, \tau_n)}\ \mathcal{T}\text{-Enum}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\ \tau_1 \qquad \Gamma, x : \tau_2 \vdash \varphi : \mathsf{Bool} \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \{x \in S : \varphi\} : \mathsf{Set}\ \omega\,(\tau_1, \tau_2)}\ \mathcal{T}\text{-SetComp-}_1$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\ \tau_1 \qquad \Gamma, x : \tau_2 \vdash \varphi : \mathsf{Bool} \qquad \Gamma, x : \tau_2 \vdash e\,(x) : \tau \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \{e\,(x) : x \in S\} : \mathsf{Set}\ \tau}\ \mathcal{T}\text{-SetComp-}_2$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash \tau_1 \lhd \tau_2}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Left-Eq}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash \tau_2 \lhd \tau_1}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Right-Eq}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \mathsf{Set}\ \tau_2 \qquad \Gamma \vdash \tau_1 \lhd \tau_2}{\Gamma \vdash e_1 \in e_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Mem}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Set}\ \tau_1 \qquad \Gamma \vdash e_2 : \mathsf{Set}\ \tau_2 \qquad \Gamma \vdash \tau_1 \lhd \tau_2}{\Gamma \vdash e_1 \subseteq e_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Subset}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Set}\ \tau_1 \qquad \Gamma \vdash e_2 : \mathsf{Set}\ \tau_2 \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash e_1 \cup e_2 : \mathsf{Set}\ \tau_1}\ \mathcal{T}\text{-Cup}$$

$$\frac{\Gamma \vdash f : \mathsf{Set}\ \tau_1 \qquad \Gamma \vdash e : \mathsf{Set}\ \tau_2 \qquad \Gamma \vdash \mathsf{dom}\,(\tau_1) \equiv \tau_2}{\Gamma \vdash f[e] : \mathsf{cod}\,(\tau_1)}\ \mathcal{T}\text{-App}$$

$$\frac{\Gamma \vdash f : \mathsf{Set}\ \tau}{\Gamma \vdash \mathsf{DOMAIN}\, f : \mathsf{Set}\ \ (\mathsf{dom}\,(\tau))}\ \mathcal{T}\text{-Dom}$$

$$\frac{\Gamma \vdash S : \mathsf{Set}\ \tau_1 \qquad \Gamma x : \tau_1' \vdash e : \tau_2 \qquad \Gamma \vdash \tau_1 \equiv \tau_1'}{\Gamma \vdash [x \in S \mapsto e] : \tau_1 \to \tau_2}\ \mathcal{T}\text{-Fcn}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}\ \mathcal{T}\text{-Plus}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 < e_2 : \mathsf{Bool}}\ \mathcal{T}\text{-Less}$$

$$\frac{\Gamma \vdash h_i : \mathsf{Str} \qquad \Gamma \vdash e_j : \tau_j \qquad |\{h_1, \ldots, h_n\}| = n \qquad i, j \in 1 \mathrel{..} n}{\Gamma \vdash [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] : [h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n]}\ \mathcal{T}\text{-Rcd}$$

$$\frac{\Gamma \vdash h_i : \mathsf{Str} \qquad \Gamma \vdash S_j : \mathsf{Set}\ \tau_j \qquad |\{h_1, \ldots, h_n\}| = n \qquad i, j \in 1 \mathrel{..} n}{\Gamma \vdash [h_1 : S_1, \ldots, h_n : S_n] : \mathsf{Set}\ [h_1 \mapsto \tau_1, \ldots, h_n \mapsto \tau_n]}\ \mathcal{T}\text{-SetRcd}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle}\ \mathcal{T}\text{-Tuple}$$

$$\frac{\Gamma \vdash S_1 : \tau_1 \quad \ldots \quad \Gamma \vdash S_n : \tau_n \qquad S_i \neq \{\}, i = 1 \mathrel{..} n}{\Gamma \vdash S_1 \times \ldots \times S_n : \mathsf{Set}\ \langle \tau_1, \ldots, \tau_n \rangle}\ \mathcal{T}\text{-Product}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \tau_1 \qquad \Gamma \vdash e_3 : \tau_2 \qquad \Gamma \vdash \tau_1 \equiv \tau_2}{\Gamma \vdash \mathsf{IF}\ e_1\ \mathsf{THEN}\ e_2\ \mathsf{ELSE}\ e_3 : \tau_1}\ \mathcal{T}\text{-ITE}$$

$$\frac{\Gamma \vdash c_i : \mathsf{Bool} \qquad \Gamma \vdash e_j : \tau_j \qquad \Gamma \vdash \tau_1 \equiv \ldots \equiv \tau_{n+1} \qquad i \in 1 \mathrel{..} n, j \in 1 \mathrel{..} (n+1)}{\Gamma \vdash \mathsf{CASE}\ c_1 \to e_1 \,\square \ldots \square\ \mathsf{OTHER}\ \to e_{n+1} : \tau_1}\ \mathcal{T}\text{-Case}$$

Transition Rules

This appendix lists the collection of rewriting rules applied during the translation. This list is not comprehensive; some trivial rules are omitted. The expression $[h_i \mapsto e_i]_{i:1..n}$ abbreviates $[h_1 \mapsto e_1, \ldots, h_n \mapsto e_n]$ and $[h_i : e_i]_{i:1..n}$ abbreviates $[h_1 : e_1, \ldots, h_n : e_n]$. The expression B abbreviates the sort *Bool*. In our translation, we assume that the function argument is always in the function domain. Moreover, just to apply the rewriting rules, we require that all expressions are assigned appropriate sorts, i.e. the type checker cannot detect any error. The default type of term x is τ. Note that the equality and equivalence in Z3 use the same symbol ı = ȷ.

Many-sorted first-order logic

$$\begin{aligned}
x \in \mathit{Set}\,\tau &\rightarrow \text{TRUE} \\
\forall x \in \{e_1, \ldots, e_n\} : p(x) &\rightarrow p(e_1) \wedge \ldots \wedge p(e_n) \qquad (x \notin FV_{1..n}) \\
\exists x \in \{e_1, \ldots, e_n\} : p(x) &\rightarrow p(e_1) \vee \ldots \vee p(e_n) \qquad (x \notin FV_{1..n}) \\
\forall x \in \{y \in S : q(x)\} : p(x) &\rightarrow \forall x : (x \in S \wedge q(x)) \Rightarrow p(x) \\
\exists x \in \{y \in S : q(x)\} : p(x) &\rightarrow \exists x : x \in S \wedge q(x) \wedge p(x) \\
\forall x \in \{e(y) : y \in S\} : p(x) &\rightarrow \forall x : (\exists y : y \in S \wedge x = e(y)) \Rightarrow p(x) \\
\exists x \in \{e(y) : y \in S\} : p(x) &\rightarrow \exists x : (\exists y : y \in S \wedge x = e(y)) \wedge p(x) \\
\forall x \in S : p(x) &\rightarrow \forall x : (x \in S) \Rightarrow p(x) \\
\exists x \in S : p(x) &\rightarrow \exists x : (x \in S) \wedge p(x)
\end{aligned}$$

where $FV_{1..n} = FV(e_1) \cup \ldots \cup FV(e_n)$.

Set theory

$$\begin{aligned}
x \in \varnothing &\rightarrow \text{FALSE} \\
x \notin S &\rightarrow \neg(x \in S) \\
x \in \{e_1, \ldots, e_n\} &\rightarrow x = e_1 \vee \ldots \vee x = e_n \\
x \in \{y \in S : p(y)\} &\rightarrow x \in S \wedge p(x)
\end{aligned}$$

$$\begin{aligned}
x \in \{e(y) : y \in S\} &\rightarrow x = e(y) \land y \in S \qquad \text{(y is fresh)}\\
S \in \text{SUBSET}\, T &\rightarrow \forall x : x \in S \Rightarrow x \in T\\
S \subseteq T &\rightarrow \forall x : x \in S \Rightarrow x \in T\\
x \in \text{UNION} S &\rightarrow \exists T : T \in S \land x \in T\\
x \in e_1 \cup e_2 &\rightarrow x \in e - 1 \lor x \in e_2\\
x \in e_1 \cap e_2 &\rightarrow x \in e_1 \land x \in e_2\\
x \in e_1 \setminus e_2 &\rightarrow x \in e_1 \land \neg(x \in e_2)
\end{aligned}$$

Instances of extensionality:

$$\begin{aligned}
S = \varnothing &\rightarrow \forall x : \neg(x \in S)\\
S = \{e_1, \ldots, e_n\} &\rightarrow \forall x : x \in S \iff x = e_1 \lor \ldots \lor x = e_n\\
S = \text{SUBSET}\, T &\rightarrow \forall x : x \in S \iff (\forall y : y \in x \Rightarrow yinT)\\
S = \text{UNION}\, T &\rightarrow \forall x : x \in S \iff (\exists y : y \in T \land x \in y)\\
S = \{x \in T : p(x)\} &\rightarrow \forall x : x \in S \iff x \in T \land p(x)\\
S = \{e(y) : y \in T\} &\rightarrow \forall x : x \in S \iff (\exists y : y \in T \land x = e(y))\\
S = T \cup U &\rightarrow \forall x : x \in S \iff x \in T \lor x \in U\\
S = T \cap U &\rightarrow \forall x : x \in S \iff x \in T \land x \in U\\
S = T \setminus U &\rightarrow \forall x : x \in S \iff x \in T \land \neg(x \in U)\\
\forall x : x \in S \iff x \in T \rightarrow S = T &
\end{aligned}$$

Functions

$$\begin{aligned}
[x \in S \mapsto e(x)][a] &\rightarrow e(a)\\
[f \text{ EXCEPT } ![x] = y][a] &\rightarrow \text{IF } x = a \text{ THEN } y \text{ ELSE } f[a]\\
\text{DOMAIN } [x \in S \mapsto e] &\rightarrow S\\
\text{DOMAIN } [f \text{ EXCEPT } ![x] = y] &\rightarrow \text{DOMAIN } f\\
f \in [S \rightarrow T] &\rightarrow \land\, isAFcn(f)\\
&\quad\ \land \text{DOMAIN } f = S\\
&\quad\ \land \forall x : x \in S \Rightarrow f[x] \in T\\
[f \text{ EXCEPT } ![a] = b] \in [S \rightarrow T] &\rightarrow \land\, isAFcn(f)\\
&\quad\ \land \text{DOMAIN } f = S\\
&\quad\ \land a \in S\\
&\quad\ \land b \in T\\
&\quad\ \land \forall x : (x \in S \ \ a) \Rightarrow f[x] \in T\\
[x \in S' \mapsto e(x)] \in [S \rightarrow T] &\rightarrow \land S' = S\\
&\quad\ \land \forall x : (x \in S) \Rightarrow e(x) \in T
\end{aligned}$$

$$\begin{aligned}
isAFcn([x \in S \mapsto e]) &\to \text{TRUE} \\
isAFcn([f \text{ EXCEPT } ![x] = y]) &\to \text{TRUE} \\
isAFcn(f) &\xrightarrow[\substack{f \in S_1 \times \ldots \times S_n \\ f \in [h_i : S_i]}]{f \in [S \to T]} \text{TRUE}
\end{aligned}$$

Instances of extensionality:

$$\begin{aligned}
f = [x \in S \mapsto e(x)] &\to \land\, isAFcn(f) \\
&\quad\ \land \text{DOMAIN } f = S \\
&\quad\ \land \forall x : x \in S \Rightarrow f[x] = e(x) \\
g = [f \text{ EXCEPT } ![a] = b] &\to \land\, isAFcn(g) \\
&\quad\ \land \text{DOMAIN } g = \text{DOMAIN } f \\
&\quad\ \land a \in \text{DOMAIN } g \Rightarrow g[a] = b \\
&\quad\ \land \forall x : x \in \text{DOMAIN } f \setminus \{a\} \\
&\qquad\qquad \Rightarrow g[x] = f[x] \\
[x \in S \mapsto e(x)] = [x \in T \mapsto d(x)] &\to \land S = T \\
&\quad\ \land \forall x : x \in S \Rightarrow e(x) = d(x)
\end{aligned}$$

Conditional Operators

$$\begin{aligned}
x \otimes \text{ IF } c \text{ THEN } t \text{ ELSE } f &\to \text{ IF } c \text{ THEN } x \otimes t \text{ ELSE } x \otimes f \\
f[\text{ IF } c \text{ THEN } t \text{ ELSE } u] &\to \text{ IF } c \text{ THEN } f[t] \text{ ELSE } f[u] \\
O_1[\text{ IF } c \text{ THEN } t \text{ ELSE } u] &\to \text{ IF } c \text{ THEN } O_1(t) \text{ ELSE } O_1(u)
\end{aligned}$$

where $\otimes$ is an infix binary TLA+ operator such as $=$, $\in$, $\Rightarrow, \land, \Leftrightarrow, +$, or $<$, and O_1 is a prefix unary TLA+ operator such as $\neg, \text{DOMAIN}, \text{SUBSET}$ or UNION.

Tuples and records

$$\begin{aligned}
\langle e_1, \ldots, e_n \rangle[i] &\to e_i \qquad \text{when } i \in 1 \ldots n \\
t \in S_1 \times \ldots \times S_n &\to \land\, isAFcn(t) \\
&\quad\ \land \text{DOMAIN } t = 1 \ldots n \\
&\quad\ \land t[1] \in S_1 \land \ldots \land t[n] \in S_n \\
[h_i \mapsto e_i]_{i:1..n}.h_j &\to e_j \qquad \text{when } j \in 1 \ldots n \\
[r \text{ EXCEPT } !.h_1 = e].h_2 &\to \text{IF } \text{“h}_1\text{”} = \text{“h}_2\text{” THEN } e \text{ ELSE } r.h_2 \\
r \in [h_i : S_i]_{i:1..n} &\to \land\, isAFcn(r) \\
&\quad\ \land \text{DOMAIN } r = \{\text{“h}_1\text{”}, \ldots, \text{“h}_\text{n}\text{”}\} \\
&\quad\ \land r[\text{“h}_1\text{”}] \in S_1 \land \cdots \land r[\text{“h}_n\text{”}] \in S_n
\end{aligned}$$

$$\begin{aligned}
[h_i \mapsto e_i]_{i:1..n} \in [f_j : S_j]_{j:1..n} &\rightarrow \bigwedge \text{“h}_\text{i}\text{”} = \text{“f}_\text{j}\text{”} \Rightarrow e_i \in S_j \qquad i, j \in 1 \mathrel{..} n \\
\text{DOMAIN } \langle\rangle &\rightarrow \emptyset \\
\text{DOMAIN } [h_i \mapsto e_i]_{i:1..n} &\rightarrow \{\text{“h}_1\text{”}, \ldots, \text{“h}_\text{n}\text{”}\} \\
\text{DOMAIN } \langle e_1, \ldots, e_n\rangle &\rightarrow 1 \mathrel{..} n \\
\text{DOMAIN } [r \text{ EXCEPT } !.h = e] &\rightarrow \text{ DOMAIN } r
\end{aligned}$$

Instances of extensionality:

$$\begin{aligned}
t = \langle e_1, \ldots, e_n\rangle &\rightarrow \land\, isAFcn(t) \\
&\quad\ \land \text{DOMAIN } t = 1 \mathrel{..} n \\
&\quad\ \land \bigwedge t[i] = e_i \quad i \in 1 \mathrel{..} n \\
T = S_1 \times \ldots S_n &\rightarrow \forall x : x \in T \Leftrightarrow \land\, isAFcn(x) \\
&\qquad\qquad\qquad\quad \land \text{DOMAIN } x = 1 \mathrel{..} n \\
&\qquad\qquad\qquad\quad \land x[1] \in S_1 \land \ldots \land x[n] \in S_n \\
r = [h_i \mapsto e_i]_{i:1..n} &\rightarrow \land\, isAFcn(r) \\
&\quad\ \land \text{DOMAIN } r = \{\text{“h}_1\text{”}, \ldots, \text{“h}_\text{n}\text{”}\} \\
&\quad\ \land r[\text{“h}_1\text{”}] = e_1 \land \ldots \land r[\text{“h}_n\text{”}] = e_n \\
x = [y \text{ EXCEPT } !.h = e] &\rightarrow \land\, isAFcn(x) \\
&\quad\ \land \text{DOMAIN } x = \text{ DOMAIN } y \\
&\quad\ \land \text{“h”} \in \text{DOMAIN } y \Rightarrow x[\text{“h”}] = e \\
&\quad\ \land \forall k : k \in \text{DOMAIN } y \setminus \{\text{“h”}\} \Rightarrow x[k] = y[k] \\
R = [h_i : S_i]_{i:1..n} &\rightarrow \forall r : r \in R \Leftrightarrow \\
&\qquad\quad \land\ isAFcn(r) \\
&\qquad\quad \land \text{DOMAIN } r = \{\text{“h}_1\text{”}, \ldots, \text{“h}_\text{n}\text{”}\} \\
&\qquad\quad \land r[\text{“h}_1\text{”}] \in S_1 \land \ldots \land r[\text{“h}_\text{n}\text{”}] \in S_n
\end{aligned}$$

Type Constraints

This appendix lists the collection of rules which the constraint generator uses. This list is not comprehensive; some trivial rules are omitted. The Γ symbol is used to denote the environment of the type checker and the $\mathcal{S}$ symbol is used to denote the set of ground types constructed from a given TLA$^+$ specification. In our translation, we assume that a function argument is always in the function domain. Notice that the equality and equivalence in Z3 use the same symbol " =". Moreover, all variables declared by the command VARIABLES in a TLA$^+$ specification are global. The operator $x \mathrel{..} y$ requires that $x \leq y$ but our constraint language cannot represent this requirement. Fortunately, since all expressions in a given TLA$^+$ specification have TLC values, this requirement is satisfied and checked by TLC.

Type grammar

$$\begin{aligned} \tau ::= & \mathsf{Bool} \mid \mathsf{Int} \mid \mathsf{Str} \mid \\ & \mathsf{Set}\ \tau \mid \tau \to \tau \mid [h \mapsto \tau] \mid \langle \tau \rangle \mid \alpha \end{aligned}$$

Abbreviations

$$\begin{aligned} \exists \alpha_{1..n} \,.\, \mathcal{C} &\triangleq \exists \alpha_1, \ldots, \alpha_n \,.\, \mathcal{C} \\ \alpha_1 \cong \ldots \cong \alpha_n &\triangleq \alpha_1 \cong \alpha_2 \land \alpha_2 \cong \alpha_3 \land \ldots \land \alpha_{n-1} \cong \alpha_n \\ \langle\!\langle \Gamma \vdash e_i : \tau_i \rangle\!\rangle_{i:1..n} &\triangleq \langle\!\langle \Gamma \vdash e_1 : \tau_1 \rangle\!\rangle \land \ldots \land \langle\!\langle \Gamma \vdash e_n : \tau_n \rangle\!\rangle \\ [h_i \mapsto e_i]_{i:1..n} &\triangleq [h_1 \mapsto e_1, \ldots, h_n \mapsto e_n] \\ [h_i : e_i]_{i:1..n} &\triangleq [h_1 : e_1, \ldots, h_n : e_n] \end{aligned}$$

Type properties

$$\begin{aligned} \mathsf{dom}\,(\tau_1 \to \tau_2) &= \tau_1 & \mathsf{dom}\,([h_i \mapsto \tau_i]_{i=1..n}) &= \mathsf{Str} \\ \mathsf{dom}\,(\langle \tau_1, \ldots, \tau_n \rangle) &= \mathsf{Int} & \mathsf{cod}\,(\tau_1 \to \tau_2) &= \tau_2 \\ \mathsf{dot}\,([h_i \mapsto \tau_i]_{i=1..n}, h_i) &= \tau_i & \mathsf{get}\,(\langle \tau_1, \ldots, \tau_n \rangle, i) &= \tau_i \end{aligned}$$

First-order logic

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash x : \tau \rangle\!\rangle &\triangleq \tau \cong \Gamma(x) \\
\langle\!\langle \Gamma \vdash \mathsf{TRUE} : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash \mathsf{FALSE} : \tau \rangle\!\rangle \\
&\triangleq \tau \cong \mathsf{Bool} \\
\langle\!\langle \Gamma \vdash e_1 \land e_2 : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e_1 \lor e_2 : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash e_1 \Rightarrow e_2 : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash e_1 \Leftrightarrow e_2 : \tau \rangle\!\rangle \\
&\triangleq \tau \cong \mathsf{Bool} \land \langle\!\langle \Gamma \vdash e_1 : \mathsf{Bool} \rangle\!\rangle \land \langle\!\langle \Gamma \vdash e_2 : \mathsf{Bool} \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash \forall x \in e \,.\, \varphi : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash \exists x \in e \,.\, \varphi : \tau \rangle\!\rangle \\
&\triangleq \land\ \tau \cong \mathsf{Bool} \\
&\quad \land\ \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash e : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \land \langle\!\langle \Gamma, x : \alpha_2 \vdash \varphi : \mathsf{Bool} \rangle\!\rangle \\
&\qquad\qquad\quad \land\ \alpha_1 \cong \alpha_2 \\
\langle\!\langle \Gamma \vdash e_1 = e_2 : \tau \rangle\!\rangle &\triangleq \land\ \tau \cong \mathsf{Bool} \\
&\quad \land\ \exists \alpha_1 \alpha_2 \,.\, \langle\!\langle \Gamma \vdash e_1 : \alpha_1 \rangle\!\rangle \land \langle\!\langle \Gamma \vdash e_2 : \alpha_2 \rangle\!\rangle \land \alpha_1 \cong \alpha_2
\end{aligned}
$$

Sets

$$
\begin{aligned}
\langle\!\langle \Gamma \vdash e_1 \in e_2 : \tau \rangle\!\rangle &\triangleq \land\ \tau \cong \mathsf{Bool} \\
&\quad \land\ \exists \alpha_1, \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash e_1 : \alpha_1 \rangle\!\rangle \land \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\ \alpha_2 \rangle\!\rangle \\
&\qquad\qquad\quad \land\ \alpha_1 \cong \alpha_2 \\
\langle\!\langle \Gamma \vdash \{\} : \tau \rangle\!\rangle &\triangleq \exists \alpha \,.\, \tau \cong \mathsf{Set}\ \alpha \\
\langle\!\langle \Gamma \vdash \{e_1, \ldots, e_n\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \ldots \alpha_n . \land \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1..n} \\
&\qquad\qquad\quad \land\ \alpha_1 \cong \ldots \cong \alpha_n \\
&\qquad\qquad\quad \land\ \tau \equiv \mathrm{Set}\ \alpha_1 \\
\langle\!\langle \Gamma \vdash \{x \in S : \varphi\} : \tau &\triangleq \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash S : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \\
&\qquad\qquad\quad \land \langle\!\langle \Gamma, x : \alpha_2 \vdash \varphi : \mathsf{Bool} \rangle\!\rangle \\
&\qquad\qquad\quad \land\ \tau \cong \mathsf{Set}\ \alpha_1 \\
&\qquad\qquad\quad \land\ \alpha_1 \cong \alpha_2 \\
\langle\!\langle \Gamma \vdash \{e : x \in S\} : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash S : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \land \langle\!\langle \Gamma, x : \alpha_1 \vdash e : \alpha_2 \rangle\!\rangle \\
&\qquad\qquad\quad \land\ \tau \cong \mathsf{Set}\ \alpha_2 \\
\langle\!\langle \Gamma \vdash \mathsf{SUBSET}\ S : \tau \rangle\!\rangle &\triangleq \exists \alpha \,.\, \langle\!\langle \Gamma \vdash S : \mathsf{Set}\ \alpha \rangle\!\rangle \land \tau \cong \mathrm{Set\ Set}\ \alpha \\
\langle\!\langle \Gamma \vdash \mathsf{UNION}\ S : \tau \rangle\!\rangle &\triangleq \exists \alpha \,.\, \tau \cong \mathsf{Set}\ \alpha \land \langle\!\langle \Gamma \vdash S : \mathsf{Set}\ \alpha \rangle\!\rangle \\
\langle\!\langle \Gamma \vdash e_1 \subseteq e_2 : \tau \rangle\!\rangle &\triangleq \land\ \tau \cong \mathsf{Bool} \\
&\quad \land\ \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \land \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\ \alpha_2 \rangle\!\rangle \\
&\qquad\qquad\quad \land\ \alpha_1 \cong \alpha_2
\end{aligned}
$$

$$\begin{aligned}\langle\!\langle \Gamma \vdash e_1 \cup e_2 : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash e_1 \cap e_2 : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash e_1 \setminus e_2 : \tau \rangle\!\rangle \\ &\triangleq \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash e_1 : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \\ &\qquad\quad \land \langle\!\langle \Gamma \vdash e_2 : \mathsf{Set}\ \alpha_2 \rangle\!\rangle \\ &\qquad\quad \land \tau \cong \mathsf{Set}\ \alpha_1 \\ &\qquad\quad \land \alpha_1 \cong \alpha_2\end{aligned}$$

Functions

$$\begin{aligned}\langle\!\langle \Gamma \vdash [x \in S \mapsto e] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash S : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \\ &\qquad\quad \land \langle\!\langle \Gamma, x : \alpha_1 \vdash e : \alpha_2 \rangle\!\rangle \\ &\qquad\quad \land \tau \cong \alpha_1 \rightarrow \alpha_2 \\ \langle\!\langle \Gamma \vdash f[e] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash f : \alpha_1 \rangle\!\rangle \\ &\qquad\quad \land \langle\!\langle \Gamma \vdash e : \alpha_2 \rangle\!\rangle \\ &\qquad\quad \land \alpha_2 \cong \mathsf{dom}\,(\alpha_1) \\ &\qquad\quad \land \tau \cong \mathsf{cod}\,(\alpha_1) \\ \langle\!\langle \Gamma \vdash \mathsf{DOMAIN}\, f : \tau \rangle\!\rangle &\triangleq \exists \alpha \,.\, \langle\!\langle \Gamma \vdash f : \alpha \rangle\!\rangle \land \tau \cong \mathsf{Set}\ (\mathsf{dom}\,(\alpha)) \\ \langle\!\langle \Gamma \vdash [S \rightarrow T] : \tau \rangle\!\rangle &\triangleq \exists \alpha_1 \alpha_2 \,.\, \land \langle\!\langle \Gamma \vdash S : \mathsf{Set}\ \alpha_1 \rangle\!\rangle \\ &\qquad\quad \land \langle\!\langle \Gamma \vdash T : \mathsf{Set}\ \alpha_2 \rangle\!\rangle \\ &\qquad\quad \land \tau \cong \mathsf{Set}\ (\alpha_1 \rightarrow \alpha_2) \\ \langle\!\langle \Gamma \vdash [f\ \mathsf{EXCEPT}\ ![a] = b] : \tau \rangle\!\rangle &\triangleq \exists \alpha_f \,.\, \land \langle\!\langle \Gamma \vdash f : \alpha_f \rangle\!\rangle \\ &\qquad\quad \land \exists \alpha_a \alpha_b \,.\, \land \langle\!\langle \Gamma \vdash a : \alpha_a \rangle\!\rangle \\ &\qquad\qquad\qquad\quad \land \langle\!\langle \Gamma \vdash b : \alpha_b \rangle\!\rangle \\ &\qquad\qquad\qquad\quad \land \alpha_f \cong \alpha_a \rightarrow \alpha_b \\ &\qquad\quad \land \tau \cong \alpha_f\end{aligned}$$

Arithmetic

$$\begin{aligned}\langle\!\langle \Gamma \vdash n : \tau \rangle\!\rangle &\triangleq \tau \cong \mathsf{Int} \quad \text{for } n \in \mathsf{Int} \\ \langle\!\langle \Gamma \vdash \mathit{Nat} : \tau \rangle\!\rangle &\triangleq \tau \cong \mathsf{Set\ Int} \\ \langle\!\langle \Gamma \vdash \mathit{Int} : \tau \rangle\!\rangle &\triangleq \tau \cong \mathsf{Set\ Int} \\ \langle\!\langle \Gamma \vdash -e : \tau \rangle\!\rangle &\triangleq \tau \cong \mathsf{Int} \land \langle\!\langle \Gamma \vdash e : \mathsf{Int} \rangle\!\rangle \\ \langle\!\langle \Gamma \vdash x + y : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash x - y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x * y : \tau \rangle\!\rangle \\ &= \langle\!\langle \Gamma \vdash x\ \mathsf{div}\ y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x\ \mathsf{mod}\ y : \tau \rangle\!\rangle \\ &\triangleq \langle\!\langle \Gamma \vdash x : \mathsf{Int} \rangle\!\rangle \land \langle\!\langle \Gamma \vdash y : \mathsf{Int} \rangle\!\rangle \land \tau \cong \mathsf{Int} \\ \langle\!\langle \Gamma \vdash x < y : \tau \rangle\!\rangle &= \langle\!\langle \Gamma \vdash x \leq y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x > y : \tau \rangle\!\rangle = \langle\!\langle \Gamma \vdash x \geq y : \tau \rangle\!\rangle \\ &\triangleq \langle\!\langle \Gamma \vdash x : \mathsf{Int} \rangle\!\rangle \land \langle\!\langle \Gamma \vdash y : \mathsf{Int} \rangle\!\rangle \land \tau \cong \mathsf{Bool}\end{aligned}$$

$$\langle\!\langle \Gamma \vdash x \mathrel{..} y : \tau \rangle\!\rangle \triangleq \langle\!\langle \Gamma \vdash x : \mathsf{Int} \rangle\!\rangle \wedge \langle\!\langle \Gamma \vdash y : \mathsf{Int} \rangle\!\rangle \wedge \tau \cong \text{Set Int}$$

Tuples and records

$$\begin{aligned}
\langle\!\langle \Gamma \vdash \langle e_1, \ldots, e_n \rangle : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n} \,.\, \wedge \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1..n} \\
&\phantom{\triangleq \exists \alpha_{1..n} \,.\,} \wedge \tau \cong \langle \tau_1, \ldots, \tau_n \rangle \\
\langle\!\langle \Gamma \vdash S_1 \times \ldots \times S_n : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n} \,.\, \wedge \langle\!\langle \Gamma \vdash S_i : \mathsf{Set}\ \alpha_i \rangle\!\rangle_{i:1..n} \\
&\phantom{\triangleq \exists \alpha_{1..n} \,.\,} \wedge \tau \cong \mathsf{Set}\ \langle \tau_1, \ldots, \tau_n \rangle \\
\langle\!\langle \Gamma \vdash [h_i \mapsto e_i]_{i:1..n} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n} \,.\, \wedge \langle\!\langle \Gamma \vdash e_i : \alpha_i \rangle\!\rangle_{i:1..n} \\
&\phantom{\triangleq \exists \alpha_{1..n} \,.\,} \wedge \tau \cong [h_i \mapsto \alpha_i]_{i:1..n} \\
\langle\!\langle \Gamma \vdash [h_i : S_i]_{i:1..n} : \tau \rangle\!\rangle &\triangleq \exists \alpha_{1..n} \,.\, \wedge \langle\!\langle \Gamma \vdash S_i : \mathsf{Set}\ \alpha_i \rangle\!\rangle_{i:1..n} \\
&\phantom{\triangleq \exists \alpha_{1..n} \,.\,} \wedge \tau \cong \mathsf{Set}\ [h_i \mapsto \alpha_i]_{i:1..n} \\
\langle\!\langle \Gamma \vdash r.h : \tau \rangle\!\rangle &\triangleq \exists \alpha \,.\, \langle\!\langle \Gamma \vdash r : \alpha \rangle\!\rangle \wedge \tau \cong \mathsf{dot}\,(\alpha, h)
\end{aligned}$$

Miscellaneous constructs

$$\begin{aligned}
\langle\!\langle \Gamma \vdash \text{“}abc\text{”} : \tau \rangle\!\rangle &\triangleq \tau \cong \mathsf{Str} \\
\langle\!\langle \Gamma \vdash \mathsf{IF}\ c\ \mathsf{THEN}\ t\ \mathsf{ELSE}\ u : \tau \rangle\!\rangle &\triangleq \wedge \langle\!\langle \Gamma \vdash c : \mathsf{Bool} \rangle\!\rangle \\
&\ \wedge \exists \alpha \,.\, \wedge \langle\!\langle \Gamma \vdash t : \alpha \rangle\!\rangle \\
&\ \wedge \langle\!\langle \Gamma \vdash u : \alpha \rangle\!\rangle \\
&\ \wedge \tau \cong \alpha
\end{aligned}$$

$$\begin{array}{lll}
\langle\!\langle \Gamma \vdash \text{CASE}\ c_1 \rightarrow e_1 & : \tau \rangle\!\rangle \triangleq & \wedge \langle\!\langle \Gamma \vdash c_i : \mathsf{Bool} \rangle\!\rangle_{i:1..n} \\
\qquad\qquad \Box \ldots & & \wedge \exists \alpha \,.\, \wedge \langle\!\langle \Gamma \vdash e_i : \alpha \rangle\!\rangle_{i:1..n+1} \\
\qquad\qquad \Box \mathsf{OTHER} \rightarrow e_{n+1} & & \qquad\ \wedge \tau \cong \alpha
\end{array}$$

Bibliography

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering.* Cambridge University Press, 2010.

[Avi03] Jeremy Avigad. Eliminating definitions and skolem functions in first-order logic. *ACM Transactions on Computational Logic (TOCL)*, 4(3):402–415, 2003.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* John Wiley & Sons, 2004.

[BBW14] Johannes Birgmeier, Aaron R Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (ctigar). In *Computer Aided Verification*, pages 831–848. Springer, 2014.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 151–165. Springer, 2007.

[BDdM+13] Clark Barrett, Morgan Deters, Leonardo de Moura, Albert Oliveras, and Aaron Stump. 6 years of smt-comp. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[BDM09] Nikolaj Bjørner and Leonardo De Moura. Z310: Applications, enablers, challenges and directions. *Constraints in Formal Verification, CFV*, 9, 2009.

[BDP89] Leo Bachmair, Nachum Dershowitz, and David A Plaisted. Completion without failure. *Resolution of equations in algebraic structures*, 2:1–30, 1989.

[BL02] Brannon Batson and Leslie Lamport. High-level specifications: Lessons from industry. In *Formal methods for components and objects*, pages 242–261. Springer, 2002.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

[BR70] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee.* NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[BR01] Thomas Ball and Sriram K Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, pages 103–122. Springer-Verlag New York, Inc., 2001.

[Bra11] Aaron R Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.

[BT07] Clark Barrett and Cesare Tinelli. Cvc3. In *Computer Aided Verification*, pages 298–302. Springer, 2007.

[Car96] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[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 *Computer Aided Verification*, pages 359–364. Springer, 2002.

[CCGR99] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.

[CDL+12] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla+ proofs. In *FM 2012: Formal Methods*, pages 147–154. Springer, 2012.

[CDLM08] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. A tla+ proof system. In G. Sutcliffe, P. Rudnicki, R. Schmidt, B. Konev, and S. Schulz, editors, *Proc. of the LPAR Workshop Knowledge Exchange: Automated Provers and Proof Assistants (KEAPPA'08)*, number 418 in CEUR Workshop Proceedings, pages 17–37, 2008.

[CDLM10] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the tla+ proof system. In *Automated Reasoning*, pages 142–148. Springer, 2010.

[CE81] Edmund M Clarke and E Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic.* Springer, 1981.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer aided verification*, pages 154–169. Springer, 2000.

[CGL94] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[CGP99] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[CK16] Sagar Chaki and Derrick Karimi. Model checking with multi-threaded ic3 portfolios. In *Verification, Model Checking, and Abstract Interpretation*, pages 517–535. Springer, 2016.

[Coh87] Anthony G. Cohn. A more expressive formulation of many sorted logic. *J. Autom. Reason.*, 3(2):113–200, June 1987.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985.

[DDM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, 2(2), 2006.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[DLP$^+$11] John R Douceur, Jacob R Lorch, Bryan Parno, James Mickens, and Jonathan M McCune. Memoir—formal specs and correctness proofs. Technical report, Technical Report MSR-TR-2011-19, Microsoft Research, 2011.

[DMB07] Leonardo De Moura and Nikolaj Bjørner. Efficient e-matching for smt solvers. In *Automated Deduction–CADE-21*, pages 183–198. Springer, 2007.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C.R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin Heidelberg, 2008.

[dMB09] L. de Moura and N. Bjorner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 45–52, Nov 2009.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[EE01] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Academic press, 2001.

[Eme08] E Allen Emerson. The beginning of model checking: A personal perspective. In *25 Years of Model Checking*, pages 27–45. Springer, 2008.

[FP91] Tim Freeman and Frank Pfenning. Refinement types for ml. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 268–277, New York, NY, USA, 1991. ACM.

[GDM09] Yeting Ge and Leonardo De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification*, pages 306–320. Springer, 2009.

[Gme15] Annu Gmeiner. *Parameterized Model Checking of Fault-Tolerant Distributed Algorithms.* PhD thesis, Vienna University of Technology, 2015.

[Gor65] Saul Gorn. Explicit definitions and linguistic dominoes. In *Systems and Computer Science, Proceedings of the Conference held at Univ. of Western Ontario*, pages 77–115, 1965.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

[HL12] Dominik Hansen and Michael Leuschel. Translating tla+ to b for validation with prob. In *Integrated Formal Methods*, pages 24–38. Springer, 2012.

[Hol97] Gerard J Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279, 1997.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):21, 2009.

[KB83] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[Kul09] Oliver Kullmann. Fundaments of branching heuristics. *Handbook of Satisfiability*, 185:205–244, 2009.

[KVW12] Igor Konnov, Helmut Veith, and Josef Widder. Who is afraid of model checking distributed algorithms? CAV Workshop (EC)$\hat{2}$, 2012.

[KVW15] Igor Konnov, Helmut Veith, and Josef Widder. Smt and por beat counter abstraction: Parameterized model checking of threshold-based distributed algorithms. In *Computer Aided Verification*, pages 85–102. Springer, 2015.

[Lam83] Leslie Lamport. What good is temporal logic? In *IFIP congress*, volume 83, pages 657–668, 1983.

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, May 1994.

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Lam11] Leslie Lamport. Byzantizing paxos by refinement, 2011. http://research.microsoft.com/en-us/um/people/lamport/pubs/web-byzpaxos.pdf.

[LB03] Michael Leuschel and Michael Butler. Prob: A model checker for b. In *FME 2003: Formal Methods*, pages 855–874. Springer, 2003.

[LMW12] Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Formal verification of pastry using tla+. In *International Workshop on the TLA+ Method and Tools*, 2012.

[LT87] Nancy A Lynch and Mark R Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151. ACM, 1987.

[LY01] Leslie Lamport and Yuan Yu. Tlc–the tla+ model checker, 2001.

[McM93] Kenneth L McMillan. *Symbolic model checking.* Springer, 1993.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.

[MP90] Zohar Manna and Amir Pnueli. A hierarchy of temporal properties (invited paper, 1989). In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 377–410. ACM, 1990.

[MS99] Joao Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence*, pages 62–74. Springer, 1999.

[MV12] Stephan Merz and Hernán Vanzetto. Automatic verification of TLA + proof obligations with SMT solvers. In *Logic for Programming, Artificial Intelligence, and Reasoning - 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, pages 289–303, 2012.

[MV14] Stephan Merz and Hernán Vanzetto. Refinement types for TLA$^+$. In Julia M. Badger and Kristin Yvonne Rozier, editors, *6th Intl. NASA Symp. Formal Methods (NFM 2014)*, volume 8430 of *LNCS*, pages 143–157, Houston, TX, U.S.A., 2014. Springer.

[New02] Michael Newman. Software errors cost us economy \$59.5 billion annually. *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[New14] Chris Newcombe. Why amazon chose tla+. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39. Springer, 2014.

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[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.

[PL12] Daniel Plagge and Michael Leuschel. Validating b, z and tla+ using prob and kodkod. In *FM 2012: Formal Methods*, pages 372–386. Springer, 2012.

[PLD$^+$11] Bryan Parno, Jacob R Lorch, John R Douceur, James Mickens, and Jonathan M McCune. Memoir: Practical state continuity for protected modules. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 379–394. IEEE, 2011.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[Ray97] Michel Raynal. A case study of agreement problems in distributed systems: non-blocking atomic commitment. In *High-Assurance Systems Engineering Workshop, 1997., Proceedings*, pages 209–214. IEEE, 1997.

[Res] Microsoft Research–Inria. Tla+ proof system.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12(1):23–41, 1965.

[SMT15] Smt-comp 2015, 2015. http://smtcomp.sourceforge.net/2015/index.shtml.

[SMT16] The satisfiability modulo theories library (smt-lib), 2016. http://smtlib.cs.uiowa.edu/logics.shtml.

[TY02] Serdar Tasiran and Yuan Yu. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the alpha 21364 microprocessor. In *Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*, 2002.

[Van14] Hernán Vanzetto. *Proof automation and type synthesis for set theory in the context of TLA+*. PhD thesis, Université de Lorraine, 2014.

[Wol14] Sebastian Wolff. Logics seminar 2014: Model checking with ic3, 2014. http://concurrency.informatik.uni-kl.de/documents/Logics_Seminar_2014/IC3.pdf.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 214–227, New York, NY, USA, 1999. ACM.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.