TLA+ Model Checking Made Symbolic

Igor Konnov, Jure Kukovec, Thanh-Hai Tran

TLA+ Model Checking Made Symbolic

IGOR KONNOV, Inria, LORIA, University of Lorraine, CNRS, Nancy, France
JURE KUKOVEC and THANH-HAI TRAN, TU Wien, Austria

TLA$^+$ is a language for formal specification of all kinds of computer systems. System designers use this language to specify concurrent, distributed, and fault-tolerant protocols, which are traditionally presented in pseudo-code. TLA$^+$ is extremely concise yet expressive: The language primitives include Booleans, integers, functions, tuples, records, sequences, and sets thereof, which can be also nested. This is probably why the only model checker for TLA$^+$ (called TLC) relies on explicit enumeration of values and states.

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

CCS Concepts: • **Theory of computation → Logic and verification**; • **Software and its engineering → Model checking**; **Specification languages**.

Additional Key Words and Phrases: Model checking, TLA$^+$, SMT

ACM Reference Format:
Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 123 (October 2019), 30 pages. https://doi.org/10.1145/3360549

ACKNOWLEDGMENTS

Supported by the Vienna Science and Technology Fund (WWTF) through project APALACHE (ICT15-103) and the Austrian Science Fund (FWF) through project PRAVDA (P27722) and Doctoral College LogiCS (W1255-N23). Experiments presented in this paper were carried out using the Grid5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations, see http://grid5000.fr.

1 INTRODUCTION

Distributed algorithms and protocols are hard to get right, especially, when they have to tolerate faults. Recent examples include Raft [Ongaro 2014] and Kafka [Gustafson 2019]. Traditionally distributed algorithms and protocols are given in pseudo-code [Attiya and Welch 2004; Lynch 1996; Raynal 2010]. One of the reasons is that distributed algorithms vary in their assumptions about the distributed system: the communication medium, system synchrony, possible faults, etc. As a result, it is hard to find a formal language that would encompass a large set of distributed algorithms, while offering flexibility and expressiveness that is required by the algorithm designers.

Authors' addresses: Igor Konnov, VeriDis, Inria, LORIA, University of Lorraine, CNRS, Nancy, 615 rue du Jardin Botanique, Villers-lès-Nancy, Meurthe-et-Moselle, 54602, France, igor.konnov@inria.fr; Jure Kukovec, jkukovec@forsyte.at; Thanh-Hai Tran, tran@forsyte.at, Institute of Logic and Computation 192/4, TU Wien, Favoritenstraße 9-11, Vienna, 1040, Austria.

2475-1421/2019/10-ART123
https://doi.org/10.1145/3360549

Two long-term projects for designing such formal languages were initiated in the 1980s: Input/Output Automata (IOA) by [Lynch and Stark 1989] and Temporal Logic of Actions (TLA) by [Lamport 1994]. As TLA was initially designed for writing mathematical proofs about algorithms, it did not offer a concrete syntax for their specification. Rather, the algorithm designers were expected to present their algorithms in first-order logic and choose a convenient interpretation. This gap was closed by [Lamport 2002] with the introduction of TLA$^+$, which offers a rich syntax for sets, functions, tuples, records, and sequences on top of first-order logic.

The TLA$^+$ toolset offers a model checker and a theorem prover: the model checker TLC enumerates states by intepreting TLA$^+$ specifications [Yu et al. 1999], whereas the theorem prover TLAPS aids the user in writing and verifying interactive proofs [Chaudhuri et al. 2010]. While progress towards proof automation in TLAPS has been made in the last years [Merz and Vanzetto 2012], writing interactive proofs is still a demanding task. Hence, the users prefer to run TLC for days, rather than writing proofs [Newcombe et al. 2015; Ongaro 2014]. TLC is an explicit-state model checker, and thus it inevitably suffers from state-space explosion. A partial remedy to this problem is to run TLC in distributed mode, in order to split the state exploration over dozens of computers.

Perhaps, the most prominent application of TLA$^+$ tools are fault-tolerant algorithms for distributed consensus. In academia, several such algorithms were specified in TLA$^+$: Paxos [Lamport et al. 2001], Disk Paxos [Gafni and Lamport 2003], Egalitarian Paxos [Moraru et al. 2013], Flexible Paxos [Howard et al. 2016], BFT [Lamport 2011], Abstract [Guerraoui et al. 2010], and Raft [Ongaro 2014]. In industry, [Newcombe et al. 2015] and [Gustafson 2019] reported on finding real bugs by checking TLA$^+$ specifications by running the TLC model checker.

Our Approach. We are developing a more efficient symbolic model checker that is powered by a satisfiability-modulo-theory (SMT) solver such as Microsoft Z3 [De Moura and Bjørner 2008]. To make the tool usable for the TLA$^+$ community, we aim at introducing as few restrictions to the language as TLC does. Hence, whenever we have a choice between an efficient SMT encoding that restricts the input and a less efficient but general SMT encoding, we choose the general one. (Indeed, we plan optimizations for the special fragments of TLA$^+$ in the future.) Similar to TLC, we make several pragmatic assumptions about the input specifications:

(1) All input parameters are fixed. Although TLA$^+$ specifications are typically parameterized, the users restrict parameters to run TLC.
(2) Reachable states and the values of the parameters are finite structures, e.g., finite sets and functions of finite domains. This is also a requirement of TLC.
(3) Following our previous work [Kukovec et al. 2018], we assume that for each variable x, there is a set of expressions $x' = e$ and $x' \in S$ that can be be treated as assignments to x'. As a consequence, the specification can be decomposed into a set of symbolic transitions.
(4) The specification is well-typeable in our type system.

The main challenge of this work comes from the expressiveness of TLA$^+$. Among basic types, it supports Booleans, integers, and uninterpreted constants. Among structured types, it supports sets, functions, tuples, records, and sequences; all of them can be arbitrarily nested in each other. Moreover, it is common to use powersets, sets of functions, and set cardinalities in TLA$^+$ specifications. Multiple techniques were developed for sets and cardinalities in SMT [Berkovits et al. 2019; Cristiá and Rossi 2016; Drăgoi et al. 2014; Kuncak et al. 2005; Tinelli et al. 2018; von Gleissenthall et al. 2016; Yessenov et al. 2010]. Although these techniques can be used to reason about some TLA$^+$ expressions, they pose various constraints on the set theory that would not easily accommodate typical TLA$^+$ specifications. [Merz and Vanzetto 2018] introduced an unsorted SMT encoding of TLA$^+$ for discharging proof obligations in TLAPS. This encoding did not scale to model checking in our preliminary experiments. Hence, we introduce a multi-sorted encoding.

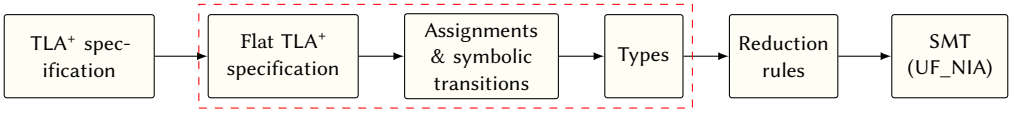

Fig. 1. The basic workflow of APALACHE. The dashed area shows the preprocessing phases.

Contributions. Our main contributions in this paper are as follows:

(1) We introduce the kernel fragment KerA$^+$ to capture all but few TLA$^+$ operators over finite structures.
(2) We define operational semantics of KerA$^+$ in terms of reduction rules. Given a KerA$^+$ formula ϕ, the reduction system produces SMT constraints that are equisatisfiable to ϕ.
(3) We prove soundness of the rewriting rules.
(4) We show how to use the reduction system for: (a) checking inductive invariants, and (b) checking safety of TLA$^+$ specifications by bounded model checking.
(5) We implement the rewriting system and run experiments on a number of benchmarks from the public TLA$^+$ repository. The experiments demonstrate that our tool APALACHE is often more efficient than TLC on the benchmarks with large state spaces.

Figure 1 shows the main phases of APALACHE [Konnov et al. 2019]. First, the call sites of user-defined operators are replaced with their bodies, which produces a flat specification. Second, the technique by [Kukovec et al. 2018] finds symbolic transitions in the specification. Third, basic type inference labels expressions with types. Finally, the reduction system produces SMT constraints. A query to the SMT solver gives us an answer to the model checking question.

Structure of the Paper. We begin with a motivating example in Section 2. We discuss the preprocessing steps in Section 3. In Section 4, we introduce the kernel language KerA$^+$. We introduce the reduction framework in Section 5 and the reduction rules in Sections 6–10. The soundness of the framework is discussed in Section 11. In Sections 12 and 13, we discuss the implementation and the experiments. We finish with the discussion of related and future work in Sections 14 and 15.

2 EXAMPLE: THE TWO-PHASE COMMIT PROTOCOL IN TLA+

A comprehensive manual on TLA$^+$ can be found in the book by [Lamport 2002]. We introduce typical TLA$^+$ constructs by discussing the famous two-phase commit protocol by [Lampson and Sturgis 1979]. In this protocol, several resource managers (e.g., databases) have to agree on whether to commit or abort a distributed transaction. The resource managers are coordinated by the transaction manager. If one of them aborts a transaction, all managers have to abort it too.

Figure 2 shows the TLA$^+$ specification of two-phase commit by [Gray and Lamport 2006]. The specification is parameterized with the set of resource managers *RM*, which, once defined, never changes in a system execution. Four variables describe the system state:

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

MODULE *TwoPhaseReformatted*

CONSTANT RM The set of resource managers (a parameter)

VARIABLES

$rmState$, $rmState[rm]$ is the state of resource manager RM

$tmState$, The state of the transaction manager

$tmPrepared$, The set of RMs from which the TM has received "Prepared" messages

$msgs$ The set of all messages sent in the distributed system

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

The transitions by the transaction manager and the resource managers:

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

The transaction manager commits the transaction:

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

The transaction manager spontaneously aborts the transaction:

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

Resource manager rm prepares:

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

Resource manager rm spontaneously decides to abort:

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

Resource manager rm is told by the TM to commit:

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

Resource manager rm is told by the TM to abort:

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

A transition of the distributed system

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

Fig. 2. The two-phase commit protocol in TLA$^+$ as specified in [Gray and Lamport 2006]. (We have only changed the indentation and comments to save some space.)

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

System transitions are defined with the operator *Next*, which is idiomatically written as a disjunction of simpler operators, called *actions*. In our example, there are two actions by the transaction manager and five actions by a resource manager. A resource manager is chosen with the existential quantifier $\exists rm \in RM$. The actions are TLA$^+$ formulas over two sets of variables: the variables without primes and the variables with primes. The former capture the state before a transition, while the latter capture the state after the transition.

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

The algorithm is designed to satisfy the following invariant:

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

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

Although this example is simple in comparison to fault-tolerant protocols such as Raft [Ongaro 2014], it demonstrates several idiosyncrasies of TLA$^+$. First, there is no fixed order of evaluating the expressions. An operator such as *Next* is just a logical formula. As soon as a vector of values for primed and non-primed variables satisfies the formula, it gives us a system transition. Second, there is no notion of an assignment. Hence, constraints on the primed variables may have different forms. Third, the language is untyped. As a result, the same variable may contain values of different types during an execution, and sets may contain type-incompatible elements.

In Section 3, we discuss how to deal with these issues before doing the translation to SMT.

3 PREPROCESSING: FLATTENING, ASSIGNMENTS, AND TYPES

3.1 Flattening

As exemplified by Section 2, TLA$^+$ specifications are normally written as a collection of operator definitions. They can be also organized in modules. As the operator *Next* describes one step of a system execution, the operators in TLA$^+$ are usually non-recursive. They are similar to macros in programming languages. As a first step, our technique replaces calls to the user-defined operators with the operator bodies; as expected, the formal arguments are substituted with the arguments at the call sites. The same applies to the local operators that are defined with the LET-IN expression. We also instantiate modules, in order to obtain a single-module specification, in which the operators *Init* and *Next* contain only the calls to the built-in TLA$^+$ operators. The flattening phase is purely syntactic, so we obviously obtain an equivalent TLA$^+$ specification.

Note on Recursive Operators. [Lamport 2018] recently added recursive operators to TLA$^+$ version 2. Hence, the users can conveniently write expressions in terms of recursion instead of logical formulas. As is common in bounded model checking, we could unroll a call to a recursive operator up to a bound predefined by the user, which would produce a large TLA$^+$ formula. To implement an incremental unrolling, we would need an advanced type checker, which we postpone for the future.

3.2 Assignments and Symbolic Transitions

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

Recently, we introduced a symbolic technique for finding such assignments without evaluating the TLA$^+$ formula [Kukovec et al. 2018]. Additionally, we proposed a technique for decomposing a TLA$^+$ formula into a disjunction of formulas $T_1, \ldots, T_k$ in the following way:

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

We apply this technique to find assignments and symbolic transitions.

Example 3.1. Consider the example in Figure 2. There are 7 symbolic transitions, corresponding to the possible actions *TMCommit*, *TMAbort*, *TMRcvPrepared*(rm), and so on. The body of *TMAbort* contains assignments to all five variables; two of them are unchanged. ◁

3.3 Types

Whereas TLA$^+$ is untyped by design, TLC dynamically computes types and rejects some combinations of legal TLA$^+$ expressions, e.g., $\{1, \text{“}a\text{”}\}$. However, TLC's type system is not defined. We use the following type system, which is similar to the type system by [Merz and Vanzetto 2012]:

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

The type system rejects some TLA$^+$ expressions that are legal in the untyped language. Importantly, elements of sets must have the same type. For example, $\{1, \{2, 3\}\}$ is ill-typed. Similarly, TLA$^+$ functions can be defined on values of different types and return values of different types, but such functions are rejected by the type system. Finally, our type system clearly distinguishes between functions, sequences, tuples, and records.

Developing a fully automatic type inference engine for TLA$^+$ is a challenge on its own. In this paper, we follow a simple approach: In most cases, the types are computed automatically by propagation; when the tool fails to find a type, it asks the user to write a type annotation. Given the syntax tree of a TLA$^+$ expression, our basic type inference algorithm works as follows:

(1) A leaf expression is assigned the respective type. For instance, the literals $0, 1, -1, \ldots$ have type Int, and the literals FALSE and TRUE have type Bool. If the type is ambiguous, as in $\{\}$, then type inference fails, and the user has to annotate the expression with a type.
(2) A non-leaf expression is an application of a built-in operator. The type signatures of these operators are predefined, e.g., $+ : \mathsf{Int} * \mathsf{Int} \to \mathsf{Int}$. Some operators introduce bound variables, e.g., $\exists x \in S : e$ or $\{e : x \in S\}$. As expected, the type of the binding set is computed first, and then the type of e is computed.

In practice, the user has only to give the types of empty sets, empty sequences, and records. It is common to mix records of different types. In Section 2, records $[type \mapsto \text{“Abort”}]$ and $[type \mapsto \text{“Prepared”}, rm \mapsto rm]$ are both added to the set *msgs*. The user has to annotate the records and their sets with a super type, e.g., $[type : \mathsf{Name}, rm : \mathsf{Set}[\mathsf{Name}]]$.

4 KERA+: THE KERNEL LANGUAGE OF TLA+ EXPRESSIONS

Our main goal is to check TLA$^+$ specifications using an SMT solver as a back-end. A direct translation of the rich TLA$^+$ syntax would be tedious and error-prone. Hence, we introduce KERA$^+$: A small set of operators that can express all but a few TLA$^+$ expressions. For example, it includes the

Table 1. The language KerA$^+$. We highlight the expressions that do not have counterparts in pure TLA$^+$.

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

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

A list of KerA$^+$ expressions is given in Table 1. It might seem surprising that very basic operators such as Boolean operators are missing. In fact, they can be expressed with IF-THEN-ELSE:

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

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

- *Assignment* $x' \in S$: Following TLC, under the conditions given by [Kukovec et al. 2018], we treat an expression $x' \in S$ as an assignment of a value from the set S to the variable x'. Note that an expression $x' = e$ is a special case of this rule, which can be written as $x' \in \{e\}$. We label such assignments with $x' \in S$, to distinguish them from membership tests $x' \in S$.
- *Non-deterministic disjunction* $\phi_1 \oplus \ldots \oplus \phi_n$: This operator formalizes the special form of TLC disjunction. It evaluates to true if and only if the disjunction $\phi_1 \vee \cdots \vee \phi_n$ evaluates to true. However, non-deterministic disjunction adds constraints on the variable assignments: For every $i, j \in 1..n$ and $i \neq j$, formula ϕ_i contains an assignment to a variable x' if and only if formula ϕ_j contains an assignment to x'. Note that this property is implied by the single-assignment property of symbolic transitions (see Section 3.2). Hence, we use it to compose the symbolic transitions.
- *Choice with an oracle* FROM $e_1, \ldots, e_n$ BY θ: This operator returns expression e_i when $\theta = i$ and $1 \leq i \leq n$; otherwise, it returns an arbitrary value of the same type as $e_1, \ldots, e_n$.

KerA$^+$ is a subset of TLA$^+$ — except for the three operators discussed above — and the meaning of the operators coincides with the description in the book by [Lamport 2002]. Denotational semantics of TLA$^+$ in first-order logic is given by [Merz 2008]. In Sections 6–10, we give a brief description of each KerA$^+$ operator along with the semantics for finite structures in terms of rewriting rules.

5 REWRITING FRAMEWORK

Our goal is to translate a KerA$^+$ expression into an equisatisfiable quantifier-free SMT formula. To this end, we introduce an abstract reduction system that allows us to iteratively transform a KerA$^+$ expression by applying reduction rules. The central idea of our approach to rewriting is to construct an overapproximation of the data structures with a graph whose edges connect values such as sets and their elements. We call this graph an *arena*, as it resembles the in-memory data structures that

are created by the explicit-state model checker TLC. While some rules for KerA$^+$ operators extend the arena with new nodes and edges, other rules use this graph to produce SMT constraints on the actual values. The reduction rules collapse a complex KerA$^+$ expression into a so-called cell that captures the result of symbolically evaluating the expression. The rewriting process terminates, when the input KerA$^+$ formula ϕ has been collapsed to a single cell. In this case, the reduction rules have produced a set of SMT constraints that are equisatisfiable to the formula ϕ.

5.1 Cells

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

New cells are introduced when rewriting a KerA$^+$ expression. For example, the expression $\{1, 2\}$ is rewritten by a series of rewriting steps: $\{1, 2\} \rightsquigarrow \{\mathrm{c}_1, 2\} \rightsquigarrow \{\mathrm{c}_1, \mathrm{c}_2\} \rightsquigarrow \mathrm{c}_3$. We give the precise definition of $\rightsquigarrow$ in Section 5.4. While the original expression does not contain cells, the rewritten expressions do. In fact, cells are well-formed KerA$^+$ expressions, as they can be seen as KerA$^+$ constants. Hence, the introduced cells can be seen as: (1) first-order constants in SMT, and (2) KerA$^+$ constants in KerA$^+$, which would be introduced in TLA$^+$ using the string notation, e.g., "abc".

5.2 Arenas

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

(1) There are no duplicate labels. Formally, for every pair $(v_1, i_1, w_1), (v_2, i_2, w_2) \in \mathcal{E}$, if $v_1 = v_2$ and $w_1 \neq w_2$, then $i_1 \neq i_2$.
(2) There are no gaps in the labels. Formally, for every $(v, i, w) \in \mathcal{E}$, and every index $j \in 1..(i-1)$, there is a cell $w \in \mathcal{V}$ with the property $(v, j, w) \in \mathcal{E}$.

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

- Notation $\mathcal{A}, \mathrm{c} : \tau$ to introduce the arena $(\mathcal{V}', \mathcal{E}')$ such that $\mathcal{V}' = \mathcal{V}(\mathcal{A}) \cup \{c\}$ and $\mathcal{E}' = \mathcal{E}(\mathcal{A})$, provided that c is a fresh cell of type τ, i.e., $\mathrm{c} \notin \mathcal{V}(\mathcal{A})$.
- Notation $\mathcal{A}, \mathrm{c} \rightarrow \mathrm{c}_1, \ldots, \mathrm{c}_n$ to introduce the arena $\mathcal{A}'$ such that $\mathcal{V}(\mathcal{A}') = \mathcal{V}(\mathcal{A})$ and $\mathcal{E}(\mathcal{A}') = \mathcal{E}(\mathcal{A}) \cup \{(\mathrm{c}, i, \mathrm{c}_i) \mid 1 \leq i \leq n\}$.

Example 5.1. Figure 3 shows examples of memory arenas for several KerA$^+$ expressions. In example (a), the arena contains six cells: three cells of type Int that represent integers 1, 2, 3; two cells of type Set[Int] that represent the sets $\{1, 2\}$ and $\{2, 3\}$; and one cell of type Set[Set[Int]] that represents the set of sets $\{\{1, 2\}, \{2, 3\}\}$. Importantly, the arena only gives us a static overapproximation of the set. The actual contents of the set encoded by cell c_6 may be $\{\emptyset\}$ or $\{\{1\}, \{2\}\}$. The further constraints on the cell contents are encoded in SMT, see Section 5.3.

In example (b), the arena contains five cells: three cells to encode the integers, the cell c_{14} to encode the record $[b \mapsto 0, c \mapsto 3]$, and the cell c_{15} to encode the tuple $\langle$"a", 3, $[b \mapsto 0, c \mapsto 3]\rangle$. In case of tuples, the cell type gives us unambiguous relation between the tuple fields and the cells pointed

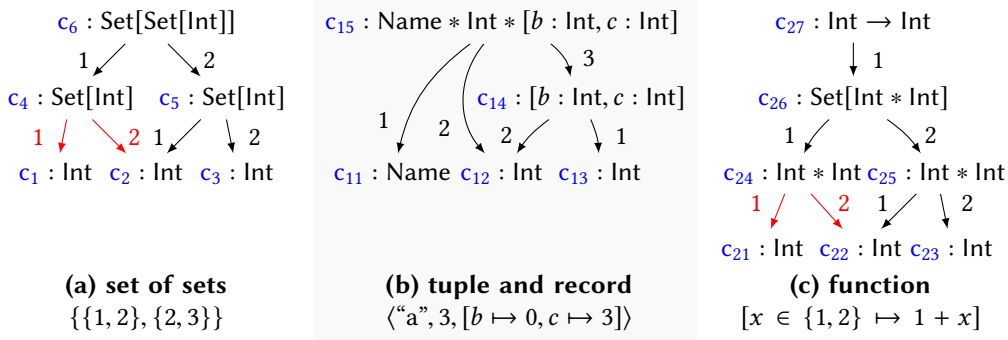

Fig. 3. Examples of arenas for data structures in KERA$^+$. The leaf cells are equal to the following constants: $c_1 = c_{21} = 1$, $c_2 = c_{22} = 2$, $c_3 = c_{23} = 3$, $c_{11} = a$, $c_{12} = 3$, and $c_{13} = 0$

by the cell. For instance, from the edge $c_{15} \xrightarrow{1} c_{11}$ and the tuple type Name $*$ Int $*$ $[b : \text{Int}, c : \text{Int}]$, we immediately obtain that cell c_{11} is the first field of the tuple c_{15}. The same applies to records.

Finally, example (c) shows the arena that is constructed for the function $f = [x \in \{1, 2\} \mapsto 1 + x]$. In our encoding, a function f is represented with its relation, that is, the set $\{(x, f[x]) : x \in \text{DOMAIN } f\}$. Hence, the cells c_{21}, c_{22}, and c_{23} encode the integers 1, 2, and 3 respectively. The cells c_{24} and c_{25} encode the pairs $\langle 1, 2\rangle$ and $\langle 2, 3\rangle$ of the relation respectively. The cell c_{26} encodes the function relation, which is pointed by the function cell c_{27}. While the function cell c_{27} may look redundant in the presence of the cell c_{26}, we keep the both, as they have different types. ◁

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

5.3 SMT constraints

We recapitulate the necessary notions related to many-sorted first-order logic. We assume fixed a set of *sorts* $\mathcal{S}$, which includes exactly one sort s_τ per type τ that is defined in Section 3. Further, let $\mathcal{F}$ be a set of *functional symbols*, each functional symbol is assigned a non-negative arity. For convenience, we say that the set of cells C coincides with the set of functional symbols of arity 0 from the set $\mathcal{F}$. Each symbol $f \in \mathcal{F}$ is assigned a sort $\mathit{sort}(f) \in \mathcal{S}$. The *ground terms* are defined as follows: (1) every constant $c \in C$ is a ground term, and (2) if $t_1, \ldots, t_n$ are ground terms and $f \in \mathcal{F}$ has arity n, then $f(t_1, \ldots, t_n)$ is a ground term, if the sorts of $f, t_1, \ldots, t_n$ are compatible.

We distinguish the set of predicates $\mathcal{P} \subseteq \mathcal{F}$, which contains the symbols that are assigned a sort $s_{\tau_1 \times \cdots \times \tau_n \to \text{Bool}}$ for $n \geq 0$ and some types $\tau_1, \ldots, \tau_n$. A *ground first-order quantifier-free formula* (FO-formula) is a Boolean combination of predicates. We assume that set $\mathcal{F}$ contains the standard symbols of integer arithmetic along with uninterpreted functions, and their interpretation is standard. In particular, the sorts s_{Bool} and s_{Int} are the sorts of Booleans and integers, respectively. The sorts for the other types are uninterpreted. Hence, we deal with the formulas of logic `QF_UFNIA` [Barrett et al. 2017]. (Integer arithmetic in TLA$^+$ does not have to be linear.)

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

Example 5.2. Consider the edge $e_{41} = (c_4, 1, c_1)$ in Figure 3 (a). If $en\langle e_{41}\rangle$ evaluates to true, then the cell c_1 belongs to the set encoded by the set c_4; otherwise, c_1 does not belong to the set. ◁

5.4 Abstract Reduction System (ARS)

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

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

We define $\rightsquigarrow$ via a set of reduction rules. For instance, the rules (Bool) and (Int) below define transitions that reduce Boolean and integer literals to cells. In the reduction rules, we write the premises above the bar and the new state of the reduction system below the bar. By convention, the state is always written as the first premise, using the notation $\langle e \mid \mathcal{A} \mid \nu \mid \Phi\rangle$.

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

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

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

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

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

$$\frac{\langle Op(e_1, \ldots, e_n) \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0\rangle \qquad \dfrac{\langle e_i \mid \mathcal{A}_{i-1} \mid \nu_{i-1} \mid \Phi_{i-1}\rangle}{\langle c_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i\rangle} \qquad \text{for } 1 \leq i \leq n}{\langle Op(c_1, \ldots, c_n) \mid \mathcal{A}_n \mid \nu_n \mid \Phi_n\rangle}\ (\textsc{RedArg})$$

To apply the reduction system to a KerA$^+$ expression e, e.g., to *Init* and *Next*, we introduce an initial state $\langle e_0 \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0\rangle$, whose arena, binding, and SMT constraints are empty. Formally, $\mathcal{A}_0 = (\emptyset, \emptyset)$, $\Phi_0 = \emptyset$, and $\nu(x) = \bot$ for $x \in$ *Vars*. Usually, the expression e_0 is a formula, that is, it has type Bool. For simplicity, we also assume that all constants that appear in e_0 have basic types, that is, Int, Bool, and Name, while the expressions of more complex types are constructed with built-in TLA$^+$ operators. This restriction is not crucial, as one can initialize TLA$^+$ parameters (called "CONSTANTS" in TLA$^+$) by evaluating an additional formula, similar to *Init*. Then, we apply the reduction rules until one of the following states is reached: (1) an *error* state, in which no rule applies, or (2) a *terminal* state, in which the expression is a single cell. If an error state has been reached, then the expression e is not well-formed.

When a terminal state c_{term} is reached, and the terminal cell c_{term} has type Bool, we add the assertion c_{term} to the SMT constraints and check their satisfiability. In Sections 6–10, we introduce rewriting rules for sets, functions, tuples, records, sequences, and control operators. Section 11 contains soundness proofs.

6 SETS

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

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

$$\frac{\langle \{c_1, \ldots, c_n\} : \mathrm{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi\rangle}{\langle c_{set} : \mathrm{Set}[\tau] \mid \mathcal{A}, c_{set}, c_{set} \to c_1, \ldots, c_n \mid \nu \mid \Phi, \bigwedge_{1\le i\le n} en\langle c_{set}, i, c_i\rangle\rangle} \text{ (Enum)}$$

Set Membership. An expression $c_x \in c_S$ such that $c_S \to_{\mathcal{A}} c_1, \ldots, c_n$ is reduced to $\bigvee_{1\le i\le n} c_x = c_i$.

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

$$\frac{\begin{array}{c}\langle \{x \in c_S : p\} : \mathrm{Set}[\tau] \mid \mathcal{A} \mid \nu \mid \Phi\rangle \\ \hline \langle p[c_1/x], \ldots, p[c_n/x] \mid \mathcal{A} \mid \Phi \mid \nu\rangle \\ \hline \langle c_1^p, \ldots, c_n^p \mid \mathcal{A}' \mid \Phi' \mid \nu'\rangle \qquad c_S \to_{\mathcal{A}} c_1, \ldots, c_n\end{array}}{\langle c_T : \mathrm{Set}[\tau] \mid \mathcal{A}', c_T \to c_1, \ldots, c_n \mid \nu' \mid \Phi', \mathit{InFilter}\rangle} \text{ (Filter)}$$

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

$$en\langle c_T, i, c_i\rangle \leftrightarrow \left(c_i^p \wedge en\langle c_S, i, c_i\rangle\right) \text{ for } 1 \le i \le n \qquad (\mathit{InFilter})$$

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

$$\frac{\begin{array}{c}\langle \mathrm{union}\ c_S : \mathrm{Set}[\mathrm{Set}[\tau]] \mid \mathcal{A} \mid \nu \mid \Phi\rangle \\ \hline c_S \to_{\mathcal{A}} c_S^1, \ldots, c_S^n \qquad c_S^i \to_{\mathcal{A}} c_1^i, \ldots c_{m_i}^i \text{ for } 1 \le i \le n\end{array}}{\langle c_U : \mathrm{Set}[\tau] \mid \mathcal{A}, c_U, c_U \to c_1^1, \ldots, c_{m_1}^1, c_1^2, \ldots, c_{m_2}^2, \ldots, c_{m_n}^n \mid \nu \mid \Phi,\ \mathit{InU}\rangle} \text{ (Union)}$$

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

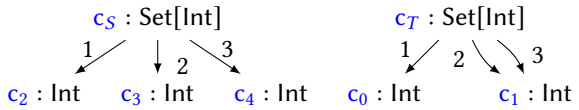

Fig. 4. An arena constructed for the set comprehension $\{x \div 3 : x \in \{2, 3, 4\}\}$. Every cell c_i has value i for $0 \leq i \leq 4$. Cell c_S encodes the set $\{2, 3, 4\}$, and cell c_T encodes the result of the set comprehension.

$$en\langle c_U, idx_{i,j}, c_j^i\rangle \leftrightarrow \left(en\langle c_S^i, j, c_j^i\rangle \wedge en\langle c_S, i, c_S^i\rangle\right) \text{ for } 1 \leq i \leq n, 1 \leq j \leq m_i, \qquad (InU)$$

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

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

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

$$\frac{\left\langle \{e : x \in c_S\} \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \text{ and } c_S \rightarrow_{\mathcal{A}} c_1, \ldots, c_n \qquad \frac{\left\langle e[c_1/x], \ldots, e[c_n/x] \mid \mathcal{A} \mid \Phi \mid \nu \right\rangle}{\left\langle c_1^e : \tau, \ldots, c_n^e : \tau \mid \mathcal{A}' \mid \Phi' \mid \nu' \right\rangle}}{\left\langle c_T \mid \mathcal{A}', c_T : \text{Set}[\tau], c_T \rightarrow c_1^e, \ldots, c_n^e \mid \nu' \mid \Phi', InMap \right\rangle} \text{(MAP)}$$

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

$$en\langle c_T, i, c_i^e\rangle \leftrightarrow en\langle c_S, i, c_i\rangle \text{ for } 1 \leq i \leq n \qquad (InMap)$$

Example 6.1. Consider Figure 4. The cell c_1 is mapped to the cell c_0, whereas the cells c_3 and c_4 are mapped to the cell c_1. Assume that the SMT solver sets $en\langle c_S, 3, c_4\rangle$ to true and $en\langle c_S, 2, c_3\rangle$ to false. Hence, $en\langle c_T, 3, c_1\rangle$ holds true and $en\langle c_T, 2, c_1\rangle$ does not. Still, c_1 belongs to the set encoded by c_T, as the edge $(c_T, 3, c_1)$ is enabled. ◁

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

Set Equality. As sets are encoded as constants of uninterpreted sorts in SMT, it is not sound to use the SMT equality. One way of imposing equality constraints is by writing down the set equality axioms as done by [Merz and Vanzetto 2018]. However, such axioms immediately introduce quantified formulas in SMT. Instead of axioms, we implement lazy equality in the rule (SETEQ).

Whenever two cells c_S and c_T are compared for the first time, (SetEq) rewrites the definition of set equality into a Boolean cell c_{eq}. Additionally, it adds the SMT constraint $c_S = c_T \leftrightarrow c_{eq}$, which allows us to use SMT equality in the later occurrences of $c_S = c_T$.

$$\frac{\langle c_S = c_T \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad \dfrac{\langle (\forall x \in c_S : x \in c_T) \wedge (\forall x \in c_T : x \in c_S) \mid \mathcal{A} \mid \nu \mid \Phi\rangle}{\langle c_{eq} : \text{Bool} \mid \mathcal{A}' \mid \nu' \mid \Phi'\rangle}}{\langle c_{eq} \mid \mathcal{A}' \mid \nu' \mid \Phi', c_S = c_T \leftrightarrow c_{eq}\rangle} \text{ (SetEq)}$$

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

$$k_0 = 0 \qquad \text{and} \qquad k_{i+1} = \text{ITE}(en\langle c_S, i, c_i\rangle \wedge notSeen_i, 1 + k_i, k_i) \text{ for } 0 < i \leq n \tag{1}$$

Equation (2) requires that the ith element contributes to the cardinality, if the previously considered elements are either outside of the set, or are different from the ith element:

$$notSeen_i = \bigwedge\nolimits_{1 \leq j < i} (en\langle c_S, j, c_j\rangle \rightarrow c_j \neq c_i) \text{ for } 0 < i \leq n \tag{2}$$

Hence, *Cardinality*(c_S) = k_n. A more efficient approach can be applied to a more restricted fragment, e.g., BAPA by [Kuncak et al. 2005]. We plan to use specialized approaches in the future.

7 PICKING SET ELEMENTS

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

The operator FROM $e_1, \ldots, e_n$ BY θ is not part of TLA$^+$. The syntax for TLA$^+$ proofs [Lamport 2018] has a similar operator PICK $x \in S$, which returns an arbitrary element of the set S. However, PICK does not provide us with fine control of which element could be picked. We define several reduction rules for FROM $e_1, \ldots, e_n$ BY θ, which vary by the types of the expressions $e_1, \ldots, e_n$.

Picking Basic Values. The rule (FromBasic) applies to Booleans, integers, and constants. It introduces a new cell c_{pick} and requires that c_{pick} equals to the θth value as prescribed by the oracle. When the oracle has a value outside of 1..n, the picked value is unconstrained.

$$\frac{\langle \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad c_1 : \tau, \ldots, c_n : \tau \qquad \tau \text{ is basic}}{\langle c_{pick} \mid \mathcal{A}, c_{pick} : \tau \mid \nu \mid \Phi, \bigwedge\nolimits_{1 \leq i \leq n} (\theta = i \rightarrow c_{pick} = c_i)\rangle} \text{ (FromBasic)}$$

Picking Sets. The second rule (FromSet) picks a set element which is itself a set. This is the most intricate rule, as it requires us to construct a set that mimics the structure of every set that is captured by the cells $c_1, \ldots, c_n$. The rule assumes that every cell c_i has the same type Set[τ] for some type τ and $1 \leq i \leq n$. Without loss of generality, we assume that every cell points to exactly the same number of cells, that is, if $c_i \rightarrow_{\mathcal{A}} c_i^1, \ldots, c_i^k$ and $c_j \rightarrow_{\mathcal{A}} c_j^1, \ldots, c_j^m$, then $k = m$. If it is not the case we can introduce additional edges by replicating the last element of the sequence, e.g., if

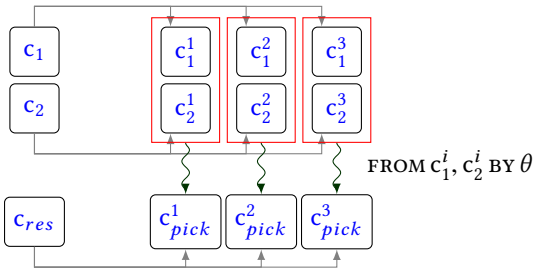

Fig. 5. Picking a set among two set cells c_1 and c_2, pointing to c^1_1, c^2_1, c^3_1 and c^1_2, c^2_2, c^3_2 respectively. The result c_{pick} points to c^1_{pick}, c^2_{pick}, c^3_{pick}, which are picked from three sequences of two cells (in red boxes).

$k < m$, then we would extend the arena as $c_i \rightarrow_{\mathcal{A}} c^1_i, \ldots, c^k_i, \ldots, c^k_i$, where c^k_i is repeated $m - k + 1$ times. (When $k = 0$, we copy the elements from the longest sequence and disable the new edges.)

The rule (FromSet) works in two steps. First, for every index $j \in 1..m$, it picks an element c^j_{pick} among the jth elements of the sets $c_1, \ldots, c_n$. Importantly, the operators FROM $c^j_1, \ldots, c^j_n$ BY θ are using the same oracle θ for every $j \in 1..m$. As a result, they pick the respective elements from the same set c_θ. Second, the resulting set c_{res} points to the picked elements $c^1_{pick}, \ldots, c^m_{pick}$.

$$\frac{\langle \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \mid \mathcal{A}_0 \mid v_0 \mid \Phi_0 \rangle \qquad \begin{array}{c} c_i : \text{Set}[\tau] \text{ for } 1 \le i \le n \text{ and } \theta : \text{Int} \\ \hline \langle \text{FROM } c^j_1, \ldots, c^j_n \text{ BY } \theta \mid \mathcal{A}_{j-1} \mid v_{j-1} \mid \Phi_{j-1} \rangle \\ \hline \langle c^j_{pick} : \tau \mid \mathcal{A}_j \mid \Phi_j \mid v_j \rangle \text{ for } 1 \le j \le m \end{array}}{c_i \rightarrow_{\mathcal{A}_0} c^1_i, \ldots, c^m_i \text{ for } 1 \le i \le n \qquad \langle c_{res} \mid \mathcal{A}, c_{res} : \text{Set}[\tau], c_{res} \rightarrow c^1_{pick}, \ldots, c^m_{pick} \mid v_m \mid \Phi_m, \mathit{InPicked} \rangle} \text{ (FromSet)}$$

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

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

Example 7.1. Figure 5 shows an example of the rule applied to FROM c_1, c_2 BY θ. The cells c_1 and c_2 have type Set[τ], each of them pointing to three element cells c^1_i, c^2_i, and c^3_i for $i \in \{1, 2\}$. The rule first applies FROM c^j_1, c^j_2 BY θ three times for $j \in \{1, 2, 3\}$ to pick one element c^j_{pick} from each pair. Note that use of θ guarantees us that the elements are drawn from the same set. The resulting cell c_{res} is pointing to the three picked cells c^1_{pick}, c^2_{pick}, and c^3_{pick}. ◁

Picking Other Values. We have also defined the rules for picking a value from: a set of functions, a set of tuples, a set of records, a set of sequences, and a powerset (constructed with SUBSET S). They are similar to (FromBasic) and (FromSet) and are omitted for brevity.

8 TUPLES AND RECORDS

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

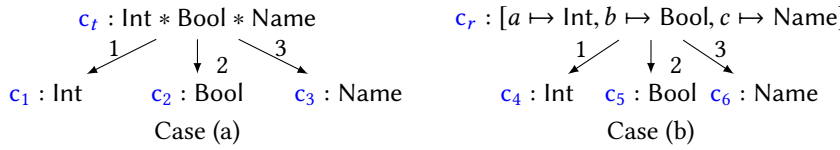

Fig. 6. (a) The arena constructed for the tuple $\langle 1, \text{TRUE}, \text{"}abc\text{"}\rangle$, assuming that the expressions 1, TRUE, and "*abc*" were rewritten into cells c_1, c_2, and c_3. (b) The arena constructed for the record $[a \mapsto 1, b \mapsto \text{TRUE}, c \mapsto \text{"}abc\text{"}]$, assuming that the expressions 1, TRUE, and "*abc*" were rewritten into cells c_4, c_5, and c_6.

Tuple Constructor. A tuple constructor adds a new cell pointing to the element cells in their index order. Figure 6 (a) shows an example of applying the rule (TupCtor).

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

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

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

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

Records. The rules for records are similar to the rules for tuples. We assume that the field names in each record type $[nm_1 \mapsto e_1, \ldots, nm_n \mapsto e_n]$ are lexicographically sorted. Obviously, there is bijection between $\{nm_1, \ldots, nm_n\}$ and $1..n$. Hence, we use the rules for tuples to rewrite most of the record operators. The only exception is DOMAIN r, which returns the set $\{nm_1, \ldots, nm_n\}$. Figure 6 (b) shows an example of rewriting a record constructor.

9 FUNCTIONS AND SEQUENCES

Functions are the second most used data structure after sets in TLA$^+$. [Lamport 2002] introduces tuples, sequences, and records as functions, so in pure TLA$^+$ any data structure different from a set is a function. As KerA$^+$ is well-typed, we treat general functions differently from tuples, records, and sequences. A function in KerA$^+$ has a type $\tau_1 \rightarrow \tau_2$, which implies that it always returns elements of the same type. Below, we define the reduction rules for function operators. In arenas, we encode a function f with its associated relation, that is, as the set of pairs $\{\langle x, f[x]\rangle : x \in \text{DOMAIN } f\}$. As a result, we reuse the rules for sets (Section 6) and tuples (Section 8). For instance, equality of two functions is simply the set equality of their associated relations.

At the arena level, a function cell c_f is always pointing to a single cell that stores the associated relation. See Figure 3 (c) for example. We use the notation $\text{funrel}(c_f)$ to refer to this relation cell.

Function Definition (FunCtor). In TLA$^+$, an expression $[x \in S \mapsto e]$ defines a function with the domain S that maps every value $v \in S$ to $e[v/x]$, where x is substituted with v in the expression e (see [Lamport 2002, p. 302]). This expression is similar to the set map $\{e : x \in S\}$. Hence, for the function constructor $[x \in S \mapsto e]$, we apply the rewriting rule (SetMap) to the expression $\{\langle x, e\rangle : x \in S\}$. This rule produces a cell c_{rel} that encodes the associated relation c_{rel} of type $\text{Set}[\tau_1 * \tau_2]$, where τ_1 is the type of elements of S, and τ_2 is the type of e. We add a cell c_f of type $\tau_1 \rightarrow \tau_2$ and make it point to c_{rel}, that is, $c_f \rightarrow_{\mathcal{A}} c_{rel}$. The rule (FunCtor) produces c_f as a result.

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

Function Update (FunExc). In TLA$^+$, an expression $[f \text{ EXCEPT } ![a] = r]$ produces a new function g that has three properties: (1) It has the same domain as f, (2) $g[x] = f[x]$ for $x \in \text{DOMAIN } f \setminus \{a\}$, and (3) $g[a] = r$ if $a \in \text{DOMAIN } f$. (See [Lamport 2002, p. 302].) Assuming that expression f has been rewritten to a cell c_f, we update the associated relation $\mathsf{funrel}(\mathrm{c}_g)$ as follows:

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

In (*Except*), all pairs that contain a as the first component are replaced with the pair $\langle a, r\rangle$, while the other pairs stay unchanged. It is easy to see that the above properties (1)-(3) are satisfied. We give the rewriting rule for ITE in Section 10.

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

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

First, the rule (FunApp) introduces an integer oracle c_{ora}, which points either to a cell from $\mathrm{c}_1, \ldots, \mathrm{c}_n$ (when $1 \le \mathrm{c}_{ora} \le n$), or an arbitrary cell of proper type (when $\mathrm{c}_{ora} = 0$). Second, a cell c_{pair} is picked using the operator $\text{FROM } \mathrm{c}_1, \ldots, \mathrm{c}_n \text{ BY } \mathrm{c}_{ora}$. This is the tuple that comprises a function argument and the respective result, so the rule (FunApp) returns the function result $\mathrm{c}_{pair}[2]$. Third, the SMT formula (*WhenInDomain*) requires the oracle to pick the right pair, that is, the one that actually belongs to the relation and whose first component is equal to the argument. Finally, the SMT formula (*WhenOutsideDomain*) allows the oracle value to be zero, only if there is no pair that matches the passed argument c_{arg}. Importantly, as the rule uses equality, we require that the lazy equality constraints $\mathrm{c}_{arg} = \mathrm{c}_i[1]$ are generated for $1 \le i \le n$.

$$\mathrm{c}_{ora} = i \rightarrow \left(\mathrm{c}_i[1] = \mathrm{c}_{arg} \wedge \mathit{en}\langle \mathrm{c}_{fun}, i, \mathrm{c}_i\rangle\right) \text{ for } 1 \le i \le n \qquad (\textit{WhenInDomain})$$

$$\mathrm{c}_{ora} = 0 \rightarrow \left(\mathrm{c}_i[1] \neq \mathrm{c}_{arg} \vee \neg \mathit{en}\langle \mathrm{c}_{fun}, i, \mathrm{c}_i\rangle\right) \text{ for } 1 \le i \le n \qquad (\textit{WhenOutsideDomain})$$

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

10 CONTROL OPERATORS AND QUANTIFIERS

Branching. The operator $\text{ITE}(\mathrm{c}_p, \mathrm{c}_1, \mathrm{c}_2)$ operator returns the value of one of its branches, depending on the Boolean condition c_p. We use the $\text{FROM } \mathrm{c}_1, \mathrm{c}_2 \text{ BY } \theta$ for $\theta \in \{1, 2\}$.

$$\frac{\left\langle \text{ITE}(\mathrm{c}_p, \mathrm{c}_1, \mathrm{c}_2) \colon \tau \mid \mathcal{A} \mid \nu \mid \Phi \right\rangle \qquad \frac{\left\langle \text{FROM } \mathrm{c}_1, \mathrm{c}_2 \text{ BY } \theta \mid \mathcal{A}, \theta : \mathsf{Int} \mid \nu \mid \Phi, 1 \le \theta \le 2 \right\rangle}{\left\langle \mathrm{c}_{res} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2 \right\rangle}}{\left\langle \mathrm{c}_{res} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2, \theta = 1 \leftrightarrow \mathrm{c}_p \right\rangle} \text{ (ITE)}$$

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

Assignments. An assignment $x' \in \mathrm{c}_S$ in KerA$^+$ specifies that a variable x' takes a value from the set S. Since any element of the set may be chosen, we use $\textsc{from}\,\mathrm{c}_1, \ldots, \mathrm{c}_n\,\textsc{by}\,\theta$ for the cells pointed by the cell c_S. We reserve the value $\theta = 0$ for the case when the set is empty, which results in assigning an arbitrary value of proper type to the variable x'.

$$\frac{\dfrac{\langle x' \in \mathrm{c}_S \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad \mathrm{c}_S \rightarrow_{\mathcal{A}} \mathrm{c}_1, \ldots, \mathrm{c}_n}{\langle \textsc{from}\,\mathrm{c}_1, \ldots, \mathrm{c}_n\,\textsc{by}\,\theta \mid \mathcal{A}, \theta : \mathsf{Int} \mid \nu \mid \Phi, 0 \le \theta \le n\rangle}}{\langle \mathrm{c} \mid \mathcal{A}_2 \mid \nu_2 \mid \Phi_2\rangle}}{\left\langle \textsc{true} \mid \mathcal{A}_2 \mid \nu_2[x \mapsto \mathrm{c}] \mid \Phi_2, \theta = 0 \leftrightarrow \bigwedge_{1 \le i \le n} \neg en\langle \mathrm{c}_S, i, \mathrm{c}_i\rangle, \bigwedge_{1 \le i \le n} (\theta \neq i \vee en\langle \mathrm{c}_S, i, \mathrm{c}_i\rangle)\right\rangle}\ (\textsc{Asgn})$$

We omit the rules for the assignments $f' \in \textsc{subset}\,S$ and $f' \in [S \rightarrow T]$ for brevity.

Substitution. A variable x can be replaced with the cell given by a valuation ν:

$$\frac{\langle x \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad x \in \mathit{Vars}}{\langle \nu(x) \mid \mathcal{A} \mid \nu \mid \Phi\rangle}\ (\textsc{Sub})$$

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

$$\frac{\langle \exists x \in \mathrm{c}_S : p \mid \mathcal{A} \mid \nu \mid \Phi\rangle \qquad \mathrm{c}_S \rightarrow_{\mathcal{A}} \mathrm{c}_1, \ldots, \mathrm{c}_n}{\langle p[\mathrm{c}_1/x] \oplus \ldots \oplus p[\mathrm{c}_n/x] \mid \mathcal{A} \mid \nu \mid \Phi\rangle}\ (\textsc{Exists})$$

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

Operator CHOOSE. By definition, $\textsc{choose}\,x \in S : p$ returns an element of S that satisfies the expression p (see [Lamport 2002, p. 294]). If there is no such an element, the result is undefined. Importantly, CHOOSE is *deterministic*: Two expressions $\textsc{choose}\,x \in S : p$ and $\textsc{choose}\,y \in T : q$ have equal values, if the filtered sets are equal, that is, $\{x \in S : p\} = \{y \in T : q\}$.

The rule CHOOSE implements this semantics as follows. First, it rewrites the set $\{x \in S : p\}$ into a cell c_F of some type τ. Suppose that c_F points to the element cells $\mathrm{c}_1, \ldots, \mathrm{c}_n$. Second, the rule applies $\textsc{from}\,\mathrm{c}_1, \ldots, \mathrm{c}_n\,\textsc{by}\,\theta$ to pick a cell c_{res} using an oracle θ. The cell c_{res} is the result of rewriting the expression $\textsc{choose}\,x \in S : p$. To guarantee determinism of CHOOSE, for each type τ, we introduce an uninterpreted function $choose_\tau$ of sort $\mathrm{Set}[\tau] \rightarrow \tau$, and require $choose_\tau(\mathrm{c}_F) = \mathrm{c}_{res}$. Finally, the rewriting system instantiates lazy equality between the pairs cells c_F^1 and c_F^2, as well as the pairs $choose_\tau(\mathrm{c}_F^1)$ and $choose_\tau(\mathrm{c}_F^2)$, which are produced by rewriting of $\{x \in S : p\}$ and $\{y \in T : q\}$

in the rule CHOOSE. Congruence of uninterpreted functions gives us the required determinism.

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

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

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

11 SOUNDNESS OF THE REDUCTION TO SMT

In this section, we define KERA$^+$ models and restrict them to finite structures. The restriction to finite structures implies that every set expression in KERA$^+$ is mapped to a finite set. Further, we present two important properties of the reduction system: termination and soundness. We introduce the invariants that are used to show soundness of the reduction. The final result guarantees that the constraints produced by the reduction system belong to the SMT theories.

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

(1) $\mathcal{D}$ is a *domain*. It is a disjoint union of sets $\mathcal{D}_1, \ldots, \mathcal{D}_n$, each $\mathcal{D}_i$ contains values of type τ_i.
(2) $\mathcal{I}$ is an *interpretation*. It assigns values from the domain to the constants and KERA$^+$ operators.

We assume that the interpretation $\mathcal{I}$ is standard, that is, it follows the standard semantics of TLA$^+$, e.g., as given by [Merz 2012]. As usual, we use the notation $[\![e]\!]^{\mathcal{M}}$ to denote the value of a KERA$^+$ expression in a model $\mathcal{M}$.

In our work, the specification parameters are fixed. Thus, every KERA$^+$ expression "intuitively" defines only finite values. We formalize this intuition by introducing finite structures and showing that every KERA$^+$ expression e defines a finite structure, as soon as the constants in e are interpreted as finite structures (see Proposition 11.1).

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

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

PROPOSITION 11.1. *Let e be a KERA$^+$ expression, and $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$ be a model. If $\mathcal{I}$ interprets all constants and free variables in e as finite structures, then the interpretation of e is a finite structure.*

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

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

Theorem 11.2. *Every sequence of ARS reductions $s_0 \rightsquigarrow s_1 \rightsquigarrow \ldots$ is finite. In other words, the reduction process terminates.*

To prove Theorem 11.2, we define a partial order on KerA$^+$ expressions and show that every reduction rule produces smaller expressions.

Theorem 11.3 formally states the soundness of our reduction system:

Theorem 11.3. *Let $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ for $1 \leq i \leq m$. Assume that e_0 is a formula, that is, it has type* Bool. *The formula e_0 is satisfiable if and only if the constraint $e_m \wedge \Phi_m$ is satisfiable.*

Note that if the reduction system terminates without an error, then the terminal expression e_m in Theorem 11.3 is a constant. Moreover, the reductions produce constraints that are compatible with SMT solvers [Barrett et al. 2017]:

Proposition 11.4. *Let $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ be a sequence of states produced by an abstract reduction system, and $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ for $1 \leq i \leq m$. Then, every formula Φ_i is a quantifier-free first-order logic formula over uninterpreted functions and integer arithmetic.*

In the following, we give the idea of our proof of Theorem 11.3. Detailed proofs are omitted. We prove the theorem by showing that the abstract reduction system satisfies six invariants on the reachable states and transitions of the system. As usual, a state s_m of the reduction system is *reachable*, if there is a finite sequence of rewriting transitions $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ from an initial state s_0 leading to s_m. Similarly, a transition is reachable, if it originates from a reachable state.

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

Invariants of the Reduction System. In order to prove soundness of the translation to SMT, we formulate six invariants on the reachable states and transitions of the abstract reduction system. Proposition 11.5 ensures that all invariants 1-6 are preserved by every sequence of transitions.

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

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

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

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

(1) Every cell c appears in either the expression e or the formula Φ if and only if it appears in $\mathcal{A}$.

(2) Arena $\mathcal{A}$ has an edge (c_{set}, i, c_{elem}) iff *the formula Φ contains the constant $en\langle c_{set}, i, c_{elem}\rangle$.*

Invariant 3 ensures that the reduction rules produce suitable models:

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

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

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

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

(2) Assume that c_{set} is a reduction of the expression FROM $c_1, \ldots, c_n$ BY θ *with* $1 \leq [\![\theta]\!]^{\mathcal{M}} \leq n$ *and* $c_{set} \rightarrow_{\mathcal{A}} c^1_{pick}, \ldots, c^m_{pick}$. *Then, the following holds* $[\![c_{set}]\!]^{\mathcal{M}} \subseteq \left\{[\![c^1_{pick}]\!]^{\mathcal{M}}, \ldots, [\![c^n_{pick}]\!]^{\mathcal{M}}\right\}$.

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

Invariant 5. *Let* $\langle e \mid \mathcal{A} \mid v \mid \Phi \rangle$ *be a reachable state of the ARS. Assume that* c_f *is a function cell of type* $\tau_1 \rightarrow \tau_2$ *in the arena* $\mathcal{A}$. *Then, there is a cell* c_{rel} *of type* $\mathrm{Set}[\tau_1 * \tau_2]$ *such that the function cell is pointing to it:* $c_f \rightarrow_{\mathcal{A}} c_{rel}$.

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

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

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

Proposition 11.5. *Let* $s_0 \rightsquigarrow \ldots \rightsquigarrow s_m$ *be a sequence of states produced by an abstract reduction system. Then, Invariant 3 is preserved by every transition* $s_i \rightsquigarrow s_{i+1}$ *for every* $0 \leq s < m$. *Moreover, Invariants 1–2, and 4–6 are preserved by every state* s_j *for every* $0 \leq j \leq m$.

12 IMPLEMENTATION

We have implemented the symbolic model checker for TLA$^+$ in Scala. It implements the stages shown in Figure 1, including the reduction rules introduced in Sections 5–10. The model checker uses the abstract syntax tree that is built by TLA$^+$ Tools — the library that contains the TLA$^+$ parser Sany and the model checker TLC. Our tool integrates with the SMT solver Z3 by [De Moura and Bjørner 2008] via the Java API. We have implemented two techniques: (1) verifying inductive invariants and (2) verifying safety with bounded model checking.

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

Recall that the formula *Next* is decomposed into a non-deterministic disjunction of symbolic transitions $T_1 \oplus \ldots \oplus T_m$ in the preprocessing phase (see Section 3). Our model checker tests Condition (2) for each transition T_i, that is, it applies the reduction system to the initial state $\langle Inv \wedge T_i \wedge \neg Inv' \mid \mathcal{A}_0 \mid v_0 \mid \Phi_0 \rangle$ and obtains the final state $\langle c^i_{final} \mid \mathcal{A}_k \mid v_k \mid \Phi_k \rangle$. The tool asks the solver, whether $\Phi_k \wedge c_{final^i}$ is satisfiable. If this is the case, the tool reports a counterexample to induction, which is obtained from the SMT model. If this is not the case for all $1 \leq i \leq m$, the inductive invariant holds true.

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

Bounded Model Checking. Given a safety property P and a number $k \geq 0$, this technique verifies, whether there is a computation of length up to k that violates the property P in one of the computation states. Equations (3)–(4) show a series of reductions that are used to encode an computation of length k. The values of the variables $\vec{x}'$ computed at step i are used as the values of the

Table 2. The list of TLA$^+$ benchmarks

Name	LOC	Description
Bakery-n	113	Bakery algorithm for mutual exclusion of n processes by Lamport
bcastByz-n	99	Reliable broadcast of n processes, f Byzantine faults by Srikanth & Toueg
bcastFolk-n	85	Folklore broadcast of n processes with f crash faults by Chandra et al.
EWD840-n	71	Termination detection in a ring of n processes by Dijkstra
Paxos-n	126	Paxos consensus (Synod) for n acceptors with crash faults by Lamport
Prisoners-n	75	Puzzle of n prisoners
Raft-n	363	Raft consensus for n processes and crash faults by Ongaro
SimpAlloc-c-r	68	Simple resource allocator with c clients and r resources by Merz
Traffic	32	Traffic example by [Wayne 2018]
TwoPhase-n	129	Two-phase commit with n resource managers by Gray & Lamport

variables $\vec{x}$ at step $i+1$. This is done by changing the variable substitution ν_i to $\nu_i[\vec{x} \mapsto \vec{x}', \vec{x}' \mapsto \bot]$.

$$\left\langle Init' \mid \mathcal{A}_0 \mid \nu_0 \mid \Phi_0 \right\rangle \rightsquigarrow^* \left\langle \mathrm{c}_1 \mid \mathcal{A}_1 \mid \nu_1 \mid \Phi_1 \right\rangle \tag{3}$$

$$\left\langle Next \mid \mathcal{A}_i \mid \nu_i[\vec{x} \mapsto \vec{x}', \vec{x}' \mapsto \bot] \mid \Phi_i \right\rangle \rightsquigarrow^* \left\langle \mathrm{c}_{i+1} \mid \mathcal{A}_{i+1} \mid \nu_{i+1} \mid \Phi_{i+1} \right\rangle \text{ for } 1 \le i \le k \tag{4}$$

To check, whether the property P can be violated after the transition $i-1$, the tool rewrites $\neg P$ as in Equation (5). Then, the SMT formula $\Phi_i^{\neg P} \wedge \mathrm{c}_i^{\neg P} \wedge \bigwedge_{1 \le j \le i} \mathrm{c}_j$ states that the property P is violated after the transition $i-1$. Satisfiability of this formula gives us a counterexample.

$$\left\langle \neg P \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \right\rangle \rightsquigarrow^* \left\langle \mathrm{c}_i^{\neg P} \mid \mathcal{A}_i^{\neg P} \mid \nu_i^{\neg P} \mid \Phi_i^{\neg P} \right\rangle \text{ for } 1 \le i \le k \tag{5}$$

13 EXPERIMENTS

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

13.1 Benchmarks

For most of our examples, we used the benchmarks from the TLA$^+$ repository of examples [TLAPlus 2019]. The traffic example is given by [Wayne 2018]. Table 2 shows the benchmarks that we use in the experiments. They range from logical puzzles to concurrent algorithms and fault-tolerant distributed algorithms. The table also lists the values of the parameters, called constants in TLA$^+$, which are used in the experiments. For each benchmark, we give the smallest reasonable value and a larger value.

These benchmarks were previously tried with TLC, some of them contain proofs of safety in TLAPS. Importantly, our modifications to the specifications are minimal. They contain type annotations and, in rare cases, equivalent expressions instead of original complex expressions that would not be handled by our tool otherwise. We neither introduced simplifications nor abstractions in the TLA$^+$ code, in order to run the model checker.

Although the repository contains 64 examples, their complexity varies. Some benchmarks are combinatorial puzzles (e.g. N-Queens, tower-of-hanoi) which are tuned to TLC, while our tool is struggling e.g. with sets of sequences, power sets, and cardinalities. We did not include about 10 trivial teaching examples (e.g. DieHard), because they are no challenge for virtually any model checker. There is a number of Paxos-like algorithms. These are rather complex TLA$^+$ specifications

Table 3. The experiments on checking inductive invariants with TLC and APALACHE.

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

Table 4. Checking candidates for inductive invariants with TLC and APALACHE that are violated.

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

of real distributed algorithms. Both TLC and our tool get stuck after 10-15 steps. We only included the famous Paxos and Raft. Some benchmarks contain recursive operators and rarely-used modules, e.g. Bags. Finally, several benchmarks are only available in the pdf format; we did not try them.

13.2 Experiments with Inductive Invariants

As explained in Section 12, APALACHE checks inductive invariants by reduction to SMT. TLC can also check inductive invariants by state enumeration. We have run both model checkers on a few benchmarks that contained inductive invariants. For each invariant, we have also introduced an invalid invariant candidate: By removing constraints, by introducing arithmetic errors, or by changing constants. This was done to check how quickly the solvers would be able to detect invariant violation as opposed to verifying the absence thereof.

Table 3 summarizes the results of the experiments with the original invariants, whereas Table 4 summarizes the results obtained when using the invalid invariant candidates. The columns "time" and "memory" show resource usage statistics. The column "#states" shows the number of distinct states explored by TLC. Finally, the columns "#tr", "#cells", and "#clauses" display the number of symbolic transitions, the number of cells in the final arena, and the number of SMT clauses introduced by APALACHE. The abbreviation 'TO' means timeout of 23 hours.

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

It was easy to check the benchmark "bcastByz" for TLC, as the inductive invariant was written for the case when no broadcast occurs in the algorithm, so the number of reachable states is just eight. Notably, TLC cannot check "Bakery" in principle, as it requires one to reason about unbounded

Table 5. The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are satisfied.

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

Table 6. The experiments on breadth-first search with TLC and bounded model checking with APALACHE. In this case, the checked safety properties are violated.

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

integers. Although APALACHE does not support infinite sets, it supports integer constants, so we added a few additional rewriting rules to handle the benchmarks like “Bakery”.

13.3 Experiments with Bounded Model Checking

Table 5 summarizes the results of the experiments with bounded model checking of safety properties. Table 6 summarizes the results of the experiments with the modified specifications that contain buggy behavior. The column “depth” shows the maximum execution length used by our tool as well as the maximum depth reached by TLC while running breadth-first search. The meaning of

the other columns is the same as in Table 3, see Section 13.2. For the small benchmarks we used the diameter bound that was reported by TLC, which does exhaustive state exploration. For the complex benchmarks we used a large enough bound on the length that allowed each experiment to finish within 24 hours. When the depth of APALACHE is smaller than the depth reported by TLC, APALACHE explores a smaller portion of the state space than TLC. For the Raft benchmark, we only report on the experiments with our tool, as TLC has produced an enormous file to store the state exploration queue and exceeded the disk quota of 100 GB in the cluster environment.

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

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

13.4 Discussion on Performance

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

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

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

$$\text{Init} \triangleq x = 0$$
$$\text{Next} \triangleq x' = 1 - x \lor x' = x$$

Bounded executions of length k of this specification pose a challenge for SMT solvers, as they often enumerate 2^k possible paths without learning. We plan to combine the presented framework with Lipton's reduction which efficiently eliminates control non-determinism, similar to the work by [Konnov et al. 2017a].

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

14 RELATED WORK

14.1 General-Purpose Specification Languages

Several general-purpose specification languages are used to specify and verify concurrent and distributed algorithms. In addition to TLA$^+$, an algorithm designer can choose from: Alloy [Jackson 2012], B [Abrial 2005], IOA [Garland and Lynch 1998], VDM [Jones 1990], or Z [Spivey and Abrial 1992]. Since these languages are widely used in both academia and industry, the conference ABZ [ABZ 2018] is held to compare and cross-fertilize these approaches. We briefly compare TLA$^+$ tools with Alloy and B. More detailed comparisons can be found in [Abbassi et al. 2018; Macedo and Cunha 2016; Newcombe 2014]. Table 7 summarizes the comparison between these languages.

Alloy [Jackson 2012] is a specification language combining relational algebra, first-order logic, transitive closures, integer arithmetic, and polymorphic types [Jackson 2012]. Alloy Analyzer uses the bounded SAT-based constraint solver Kodkod [Torlak and Jackson 2007] as its back-end. Alloy and Alloy Analyzer have been used for finding bugs in distributed algorithms, e.g., in the Chord ring membership protocol [Zave 2012].

Alloy is expressive enough to specify our benchmarks. To reduce human efforts in the encoding, we could develop a translator from TLA$^+$ to Alloy. Such work would require a set of translation rules, like the ones in our abstract reduction system. However, the translation is not straightforward because of considerable differences in two languages. For instance, Alloy does not support temporal operators, and therefore, it requires the user to use particular idioms to describe behavioral properties of systems. The analysis tools also differ. Alloy Analyzer requires the user to give precise bounds on the domain. In particular, integers have fixed bit-width, which may result in missing a counterexample. Our model checker mitigates this limitation by using SMT solvers, which reason about unbounded integers (similar to Alloy, we bound the sets). There have been a few attempts to support Alloy with SMT solvers [El Ghazi and Taghdiri 2011; Meng et al. 2017] and the tools for other languages [Krings et al. 2018; Macedo et al. 2016].

B [Abrial 2005] is a state-based specification language rooted in predicate logic, ZF set theory, types, and arithmetic. The B method has been also used in many industrial projects, e.g. railway systems [Behm et al. 1999]. Its automated analysis toolset called ProB [Leuschel and Butler 2008] supports both explicit-state model checking and constraint solving by using SICStus Prolog [Carlsson et al. 1988]. Several tools translate fragments of B to other specification languages, e.g. to TLA$^+$ [Hansen and Leuschel 2012; Plagge and Leuschel 2012] and Alloy [Krings et al. 2018]. While TLA$^+$ supports arbitrary sets, B requires that every member in a set has the same type. Our work is close to the translation by [Hansen and Leuschel 2012] from TLA$^+$ to B. This translation allows one to apply the model checker ProB, which uses constraint solving. As B and TLA$^+$ are much closer than SMT and TLA$^+$, their translation is conceptually simpler, though the authors had to deal with a few language incompatibilities. [Hansen and Leuschel 2012] report on the experiments with `SimpAlloc`, as well as with simpler benchmarks, where ProB was shown to be more efficient than TLC. We are not aware of applying this tool to fault-tolerant distributed algorithms such as Paxos, reliable broadcast "bcastByz", or two phase commit "TwoPhase".

14.2 Interactive Theorem Provers and SMT

[Merz and Vanzetto 2018] introduced two encodings to translate TLA$^+$ to SMT formulas: an untyped one and a multi-sorted one. Their work is designed towards proving unsatisfiablity of obligations inside the TLA Proof System [Chaudhuri et al. 2010]. These obligations are typically small in comparison to a complete TLA$^+$ specification, and their techniques utilize quantified formulas which are supported by SMT fairly well for the unsatisfiable case. If SMT solvers cannot decide

Table 7. General-purpose specification languages and automatic analysis tools

	TLA$^+$	**Alloy**	**B**
Theories	Untyped FOL, ZFC set theory	Typed FOL, relational algebra	Typed FOL, ZF set theory
Automatic analysis tools	TLC, APALACHE	Kodkod	ProB
Back-end solvers	SMT	SAT	SICStus Prolog

on satisfiability, the user has to prove the obligation manually. In contrast, our tool supports automatic verification. We first tried to use the untyped encoding for bounded model checking, but the search space of Z3 was significantly larger even for small examples than in the case of a multi-sorted encoding. While our type system is similar to one in [Merz and Vanzetto 2018], our abstract reduction system applies a quantifier-free encoding, and unrolls a complete TLA$^+$ specification up to k steps. This allows us to check satisfiability (when finding enabled transitions or counterexamples) as well as unsatisfiability (when proving that a transition is disabled and an invariant holds true).

Sledgehammer is a tool to combine the interactive theorem prover Isabelle [Nipkow et al. 2002] with a variety of automatic theorem provers (ATPs) and SMT solvers [Blanchette et al. 2013; Paulson and Susanto 2007]. Since Isabelle is designed for polymorphic high-order logic, the translation meets challenges in high-order features and type information. Moreover, Sledgehammer's success rate depends on lemmas extracted from Isabelle's libraries by a relevance filter, and on heuristics to instantiate quantifiers, e.g. weights and triggers.

SMTCoq [Ekici et al. 2017] is a plug-in for integrating SMT and SAT solvers into the interactive theorem prover Coq [Bertot and Castéran 2013]. The primary use case for SMTCoq aims at increasing the level of automation in Coq. SMTCoq provides tactics to translate a Coq goal into SMT expressions that use uninterpreted functions, linear integer arithmetic, bit vectors, and functional arrays. When the SMT solver produces a proof certificate, SMTCoq validates the certificate and generates a Coq proof for the original goal.

Several projects on proving correctness of distributed algorithms with interactive theorem provers were conducted by [Hawblitzel et al. 2017], [Wilcox et al. 2015], [Rahli et al. 2017], [Sergey et al. 2018], [Azmy et al. 2018], and [von Gleissenthall et al. 2019]. Although, guarantees provided by such proofs are much stronger, they demand a different level of verification efforts.

14.3 Semi-Automated Provers using Decision Procedures

[Padon et al. 2017] checked safety of several variants of Paxos in the effectively-propositional fragment of uninterpreted first-order logic (EPR). In their approach, the user specifies the transition system in first-order logic by means of uninterpreted relations and constants. The tool aids the user in interactive discovery of inductive invariants. Further, in order to fit the verification problem in EPR, the user has to come up with so-called derived relations. This is a powerful method that can be used for parameterized verification. However, the user has to invest more efforts in expressing the algorithms in uninterpreted first-order logic and interacting with the tool.

[Barnett et al. 2005; Leino 2008] developed the intermediate verification language Boogie, which serves as a layer on which to build program verifiers for other languages, e.g. VCC [Cohen et al. 2009], Dafny [Leino 2010], and Spec# [Barnett et al. 2004]. Boogie expressions are translated to the input languages of automatic theorem provers, primarily to the SMT solver Z3, by applying Hoare logic [Hoare 1969]. This approach brings a higher degree of automation, but does not eliminate the

human proof effort required since Boogie uses undecidable theories of SMT. The main application of Dafny is verification of sequential programs, whereas TLA$^+$ is built around non-determinism.

[Swamy et al. 2016] designed the general-purpose functional programming language F* with effects aimed at program verification. Like Boogie, this language utilizes SMT solvers as back-end provers, and supports interactive proofs. This language targets to fill in the gap between implementation and verification.

14.4 Model Checkers for Specialized Languages

Promela is the input language of the model checker Spin [Holzmann 2003]. Promela supports Boolean and integer variables, arrays, processes, message channels, arithmetic, and temporal operators. Spin is an explicit-state model checker that was applied to several industrial problems. Moreover, [Delzanno et al. 2014] checked a version of Paxos and [Zave 2015] checked Chord. While we could encode the benchmarks in Promela, this work requires serious efforts, as specifications in Promela are low-level in comparison to TLA$^+$. NuSMV [Cimatti et al. 2002] and nuXMV [Cavada et al. 2014] stem from the symbolic model checker SMV [McMillan 1993]. They are designed for modeling finite-state hardware protocols. The SMV language is much more restrictive than TLA$^+$.

Several techniques and tools for parameterized verification of fault-tolerant distributed algorithms were introduced by [Drăgoi et al. 2014; Drăgoi et al. 2016], [Farzan et al. 2016], [von Gleissenthall et al. 2016], [Konnov et al. 2017b], and [Maric et al. 2017]. The efficiency of these techniques comes from the restriction to special domains, whereas our approach applies to virtually any TLA$^+$ specification over finite structures.

Finally, symbolic model checking has been applied in many different application domains. For example, TAMARIN [Meier et al. 2013] focuses on security protocols, and Kind 2 [Champion et al. 2016] is designed for the dataflow language Lustre.

15 CONCLUSIONS

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

Our work is the first step towards developing an efficient symbolic model checker for TLA$^+$. Indeed, many reduction rules can be optimized for fragments of TLA$^+$. For instance, we could write more efficient rules for functions with linearly ordered domains such as integers, or rules for comparing set cardinalities to integers [Berkovits et al. 2019; Kuncak et al. 2005]. More importantly, our framework opens the door for applying more advanced techniques such as abstraction [Ball et al. 2001; Clarke et al. 2003] and reduction [Cohen and Lamport 1998; Lipton 1975]. Reductions were shown to be efficient for special classes of fault-tolerant distributed algorithms by [Damian et al. 2019; Konnov et al. 2017b; von Gleissenthall et al. 2019]. We are going to explore similar techniques, in order to check complex TLA$^+$ specifications of Raft by [Ongaro 2014], Disk Paxos [Gafni and Lamport 2003], and Egalitarian Paxos by [Moraru et al. 2013].

REFERENCES

Ali Abbassi, Amin Bandali, Nancy Day, and Jose Serna. 2018. A Comparison of the Declarative Modelling Languages B, Dash, and TLA+. In *2018 IEEE 8th International Model-Driven Requirements Engineering Workshop (MoDRE)*. IEEE, 11–20.

Jean-Raymond Abrial. 2005. *The B-book: assigning programs to meanings*. Cambridge University Press.

ABZ. 2018. 6th International ABZ Conference ASM, Alloy, B, TLA, VDM, Z, 2018.

Hagit Attiya and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics, Second Edition*. John Wiley & Sons, Inc.

Noran Azmy, Stephan Merz, and Christoph Weidenbach. 2018. A machine-checked correctness proof for Pastry. *Sci. Comput. Program.* 158 (2018), 64–80.

Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. 2001. Automatic Predicate Abstraction of C Programs. In *PLDI*. 203–213.

Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. 2005. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*. Springer, 364–387.

Mike Barnett, K Rustan M Leino, and Wolfram Schulte. 2004. The Spec# programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*. Springer, 49–69.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6*. Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. 1999. METEOR: A successful application of B in a large project. In *International Symposium on Formal Methods*. Springer, 369–387.

Idan Berkovits, Marijana Lazic, Giuliano Losa, Oded Padon, and Sharon Shoham. 2019. Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In *CAV*. 245–266.

Yves Bertot and Pierre Castéran. 2013. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media.

Jasmin Christian Blanchette, Sascha Böhme, and Lawrence C Paulson. 2013. Extending Sledgehammer with SMT solvers. *Journal of automated reasoning* 51, 1 (2013), 109–128.

Mats Carlsson, Johan Widen, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, and Thomas Sjöland. 1988. *SICStus Prolog user's manual*. Vol. 3. Swedish Institute of Computer Science Kista, Sweden.

Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. 2014. The nuXmv symbolic model checker. In *International Conference on Computer Aided Verification*. Springer, 334–342.

Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. 2016. The Kind 2 model checker. In *International Conference on Computer Aided Verification*. Springer, 510–517.

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. 2010. The TLA$^+$ proof system: Building a heterogeneous verification platform. In *Theoretical aspects of computing*. Springer-Verlag, 44–44.

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. 2002. Nusmv 2: An opensource tool for symbolic model checking. In *International Conference on Computer Aided Verification*. Springer, 359–364.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2003. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50, 5 (2003), 752–794.

Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 2009. VCC: A practical system for verifying concurrent C. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 23–42.

Ernie Cohen and Leslie Lamport. 1998. Reduction in TLA. In *CONCUR (LNCS)*. 317–331.

Maximiliano Cristiá and Gianfranco Rossi. 2016. A Decision Procedure for Sets, Binary Relations and Partial Functions. In *CAV*. 179–198.

Andrei Damian, Cezara Dragoi, Alexandru Militaru, and Josef Widder. 2019. Communication-Closed Asynchronous Protocols. In *CAV*. 344–363.

Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *TACAS*. LNCS, Vol. 1579. 337–340.

Giorgio Delzanno, Michele Tatarek, and Riccardo Traverso. 2014. Model Checking Paxos in Spin. In *Proceedings Fifth International Symposium on Games, Automata, Logics and Formal Verification, GandALF 2014, Verona, Italy, September 10-12, 2014*. 131–146.

Cezara Drăgoi, Thomas A. Henzinger, Helmut Veith, Josef Widder, and Damien Zufferey. 2014. A Logic-based Framework for Verifying Consensus Algorithms. In *VMCAI (LNCS)*, Vol. 8318. 161–181.

Cezara Drăgoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*. 400–415.

Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. 2017. SMTCoq: A plug-in for integrating SMT solvers into Coq. In *International Conference on Computer Aided Verification*. Springer, 126–133.

Aboubakr Achraf El Ghazi and Mana Taghdiri. 2011. Relational reasoning via SMT solving. In *International Symposium on Formal Methods*. Springer, 133–148.

Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving Liveness of Parameterized Programs. In *LICS*. 185–196.

Eli Gafni and Leslie Lamport. 2003. Disk Paxos. *Distributed Computing* 16, 1 (2003), 1–20.

Stephen J Garland and Nancy A Lynch. 1998. *The IOA language and toolset: Support for designing, analyzing, and building distributed systems*. Technical Report. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science.

Jim Gray and Leslie Lamport. 2006. Consensus on transaction commit. *ACM Trans. Database Syst.* 31, 1 (2006), 133–160.

Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2010. The next 700 BFT protocols. In *Proceedings of the 5th European conference on Computer systems*. ACM, 363–376.

Jason Gustafson. 2019. Kafka Improvement Proposal 320. https://cwiki.apache.org/confluence/display/KAFKA/KIP-320%3A+Allow+fetchers+to+detect+and+handle+log+truncation

Dominik Hansen and Michael Leuschel. 2012. Translating TLA + to B for Validation with ProB. In *IFM*. 24–38.

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2017. IronFleet: Proving Safety and Liveness of Practical Distributed Systems. *Commun. ACM* 60, 7 (June 2017), 83–92.

Charles Antony Richard Hoare. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10 (1969), 576–580.

Gerard Holzmann. 2003. *The SPIN Model Checker*. Addison-Wesley.

Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum Intersection Revisited. In *OPODIS*. 25:1–25:14.

Daniel Jackson. 2012. *Software Abstractions: logic, language, and analysis*. MIT press.

Cliff B Jones. 1990. *Systematic software development using VDM*. Vol. 2. Prentice Hall Englewood Cliffs.

Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. 2019. APALACHE Model Checker. https://github.com/konnov/apalache.

Igor Konnov, Marijana Lazic, Helmut Veith, and Josef Widder. 2017a. Para^2: Parameterized Path Reduction, Acceleration, and SMT for Reachability in Threshold-Guarded Distributed Algorithms. *Formal Methods in System Design* 51, 2 (2017), 270–307.

Igor Konnov, Marijana Lazić, Helmut Veith, and Josef Widder. 2017b. A Short Counterexample Property for Safety and Liveness Verification of Fault-tolerant Distributed Algorithms. In *POPL*. 719–734.

Sebastian Krings, Joshua Schmidt, Carola Brings, Marc Frappier, and Michael Leuschel. 2018. A Translation from Alloy to B. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 71–86.

Jure Kukovec, Thanh-Hai Tran, and Igor Konnov. 2018. Extracting Symbolic Transitions from TLA+ Specifications. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. 89–104.

Viktor Kuncak, Huu Hai Nguyen, and Martin C. Rinard. 2005. An Algorithm for Deciding BAPA: Boolean Algebra with Presburger Arithmetic. In *CADE*. 260–277.

Leslie Lamport. 1994. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.* 16, 3 (1994), 872–923.

Leslie Lamport. 2002. *Specifying systems: The TLA+ language and tools for hardware and software engineers*. Addison-Wesley.

Leslie Lamport. 2011. Byzantizing Paxos by Refinement. In *DISC (LNCS)*, Vol. 6950. Springer, 211–224.

Leslie Lamport. 2018. TLA^{+2}: A Preliminary Guide. https://lamport.azurewebsites.net/tla/tla2-guide.pdf

Leslie Lamport et al. 2001. Paxos made simple. *ACM Sigact News* 32, 4 (2001), 18–25.

Butler Lampson and Howard E Sturgis. 1979. Crash recovery in a distributed data storage system. (1979).

K Rustan M Leino. 2008. This is boogie 2. *manuscript KRML* 178, 131 (2008), 9.

K Rustan M Leino. 2010. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 348–370.

Michael Leuschel and Michael Butler. 2008. ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer* 10, 2 (2008), 185–203.

Richard J. Lipton. 1975. Reduction: A Method of Proving Properties of Parallel Programs. *Commun. ACM* 18, 12 (1975), 717–721.

Nancy A Lynch. 1996. *Distributed algorithms*. Morgan Kaufmann.

Nancy A. Lynch and Eugene W. Stark. 1989. A Proof of the Kahn Principle for Input/Output Automata. *Inf. Comput.* 82, 1 (1989), 81–92.

Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. 2016. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 373–383.

Nuno Macedo and Alcino Cunha. 2016. Alloy meets TLA+: An exploratory study. *arXiv preprint arXiv:1603.03599* (2016).

Ognjen Maric, Christoph Sprenger, and David A. Basin. 2017. Cutoff Bounds for Consensus Algorithms. In *CAV*. 217–237.

Kenneth L McMillan. 1993. The SMV system. In *Symbolic Model Checking*. Springer, 61–85.

Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. 2013. The TAMARIN prover for the symbolic analysis of security protocols. In *International Conference on Computer Aided Verification*. Springer, 696–701.

Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. 2017. Relational constraint solving in SMT. In *International Conference on Automated Deduction*. Springer, 148–165.

Stephan Merz. 2008. The Specification Language TLA$^+$. In *Logics of Specification Languages*, Dines Bjørner and Martin C. Henson (Eds.). Springer, Berlin-Heidelberg, 401–451.

Stephan Merz. 2012. On the Logic of TLA$^+$. *Computing and Informatics* 22, 3-4 (2012), 351–379.

Stephan Merz and Hernán Vanzetto. 2012. Automatic Verification of TLA$^+$ Proof Obligations with SMT Solvers.. In *LPAR*, Vol. 7180. Springer, 289–303.

Stephan Merz and Hernán Vanzetto. 2018. Encoding TLA+ into unsorted and many-sorted first-order logic. *Science of Computer Programming* 158 (2018), 3–20.

Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *SOSP*. ACM, 358–372.

Chris Newcombe. 2014. Why amazon chose TLA+. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Springer, 25–39.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon web services uses formal methods. *Comm. ACM* 58, 4 (2015), 66–73.

Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. 2002. *Isabelle/HOL: a proof assistant for higher-order logic*. Vol. 2283. Springer Science & Business Media.

Diego Ongaro. 2014. *Consensus: Bridging theory and practice*. Ph.D. Dissertation. Stanford University.

Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. 2017. Paxos made EPR: decidable reasoning about distributed protocols. *PACMPL* 1, OOPSLA (2017), 108:1–108:31.

Lawrence C Paulson and Kong Woei Susanto. 2007. Source-level proof reconstruction for interactive theorem proving. In *International Conference on Theorem Proving in Higher Order Logics*. Springer, 232–245.

Daniel Plagge and Michael Leuschel. 2012. Validating B, Z and TLA+ using ProB and Kodkod. In *International Symposium on Formal Methods*. Springer, 372–386.

Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable. 2017. EventML: Specification, verification, and implementation of crash-tolerant state machine replication systems. *Sci. Comput. Program.* 148 (2017), 26–48.

Michel Raynal. 2010. *Communication and Agreement Abstractions for Fault-Tolerant Asynchronous Distributed Systems*. Morgan & Claypool Publishers.

Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2018. Programming and proving with distributed protocols. *PACMPL* 2, POPL (2018), 28:1–28:30.

J Michael Spivey and JR Abrial. 1992. *The Z notation*. Prentice Hall Hemel Hempstead.

Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, et al. 2016. Dependent types and multi-monadic effects in F. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 256–270.

Cesare Tinelli, Andrew Reynolds, Clark Barrett, and Kshitij Bansal. 2018. Reasoning with Finite Sets and Cardinality Constraints in SMT. *Logical Methods in Computer Science* 14 (2018).

TLAPlus. 2019. A collection of TLA+ specifications of varying complexities. https://github.com/tlaplus/Examples

Emina Torlak and Daniel Jackson. 2007. Kodkod: A relational model finder. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 632–647.

Klaus von Gleissenthall, Nikolaj Bjørner, and Andrey Rybalchenko. 2016. Cardinalities and universal quantifiers for verifying parameterized systems. In *PLDI*. 599–613.

Klaus von Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. 2019. Pretend synchrony: synchronous verification of asynchronous distributed programs. *PACMPL* 3, POPL (2019), 59:1–59:30.

Hillel Wayne. 2018. *Practical TLA+*. Apress.

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *PLDI*. 357–368.

Kuat Yessenov, Ruzica Piskac, and Viktor Kuncak. 2010. Collections, Cardinalities, and Relations. In *VMCAI*. 380–395.

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. 1999. Model checking TLA$^+$ specifications. In *Correct Hardware Design and Verification Methods*. Springer, 54–66.

Pamela Zave. 2012. Using lightweight modeling to understand Chord. *ACM SIGCOMM Computer Communication Review* 42, 2 (2012), 49–57.

Pamela Zave. 2015. A practical comparison of Alloy and Spin. *Formal Aspects of Computing* 27, 2 (2015), 239–253.