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:

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.

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.

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.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART123
https://doi.org/10.1145/3360549
The TLA+ toolset offers a model checker and a theorem prover: the model checker TLC enumerates states by interpreting 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 power sets, sets of functions, and set cardinalities in TLA+ specifications. Multiple techniques were developed for sets and cardinalities in SMT [Berkovits et al. 2019; Cristià 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.

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 $\phi$, the reduction system produces SMT constraints that are equisatisfiable to $\phi$.
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 ⊆ 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 ↦ “Commit”], [type ↦ “Abort”], and [type ↦ “Prepared”, rm ↦ S]. The records of the third kind have an extra field rm containing a set S ⊆ RM of resource managers.

The initial system states are defined by the operator Init. This operator requires tmState to be equal to “init”, the sets tmPrepared and msgs to be empty, and rmState to be a function that constrains every resource manager rm ∈ 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
MODULE TwoPhaseReformatted

CONSTANT RM
  The set of resource managers (a parameter)

VARIABLES
  rmState,  \[ \text{rmState}[\text{rm}] \text{ is the state of resource manager RM} \]
  tmState,  \[ \text{The state of the transaction manager} \]
  tmPrepared,  \[ \text{The set of RMs from which the TM has received “Prepared” messages} \]
  msgs  \[ \text{The set of all messages sent in the distributed system} \]

Init $\overset{\Delta}{=} \quad \land \text{rmState} = [\text{rm} \in \text{RM} \mapsto \text{“working”}]$  \[ \text{constraints on the initial states} \]
$\quad \land \text{tmState} = \{ \text{“init”} \odot \text{tmPrepared} = \{\} \odot \text{msgs} = \{\} \]

The transitions by the transaction manager and the resource managers:

$\text{TMRcvPrepared(}\text{rm}\text{)} \overset{\Delta}{=} \quad \land \text{tmState} = \{ \text{“init”} \odot \text{tmPrepared} = \{\} \odot \text{msgs} \}
\quad \land \text{rmPrepared}' = \text{tmPrepared} \cup \{ \text{rm} \} \odot \text{UNCHANGED } \langle \text{rmState}, \text{tmState}, \text{msgs} \rangle$

The transaction manager commits the transaction:

$\text{TMCommit} \overset{\Delta}{=} \quad \land \text{tmState} = \{ \text{“init”} \odot \text{tmPrepared} = \{\} \odot \text{msgs} \}
\quad \land \text{msgs}' = \text{msgs} \cup \{ \text{type } \mapsto \text{“Commit”} \} \odot \text{UNCHANGED } \langle \text{rmState}, \text{tmPrepared} \rangle$

The transaction manager spontaneously aborts the transaction:

$\text{TMAbort} \overset{\Delta}{=} \quad \land \text{tmState} = \{ \text{“init”} \odot \text{tmPrepared} = \{\} \odot \text{msgs} \}
\quad \land \text{msgs}' = \text{msgs} \cup \{ \text{type } \mapsto \text{“Abort”} \} \odot \text{UNCHANGED } \langle \text{rmState}, \text{tmPrepared} \rangle$

Resource manager \text{rm} prepares:

$\text{RMPrep(}\text{rm}\text{)} \overset{\Delta}{=} \quad \land \text{rmState} = \{ \text{“working”} \}
\quad \land \text{rmState}' = [\text{rmState EXCEPT } !\text{rm} = \{ \text{“prepared”} \}]
\quad \land \text{msgs}' = \text{msgs} \cup \{ \text{type } \mapsto \text{“Prepared”}, \text{rm} \mapsto \text{rm} \}
\quad \land \text{UNCHANGED } \langle \text{tmState}, \text{tmPrepared} \rangle$

Resource manager \text{rm} spontaneously decides to abort:

$\text{RMChoosetAbort(}\text{rm}\text{)} \overset{\Delta}{=} \quad \land \text{rmState} = \{ \text{“working”} \}
\quad \land \text{rmState}' = [\text{rmState EXCEPT } !\text{rm} = \{ \text{“aborted”} \}]
\quad \land \text{UNCHANGED } \langle \text{tmState}, \text{tmPrepared}, \text{msgs} \rangle$

Resource manager \text{rm} is told by the TM to commit:

$\text{RMCvCommitMsg(}\text{rm}\text{)} \overset{\Delta}{=} \quad \land \text{type } \mapsto \text{“Commit”} \odot \text{msgs} \}
\quad \land \text{rmState}' = [\text{rmState EXCEPT } !\text{rm} = \{ \text{“committed”} \}]
\quad \land \text{UNCHANGED } \langle \text{tmState}, \text{tmPrepared}, \text{msgs} \rangle$

Resource manager \text{rm} is told by the TM to abort:

$\text{RMCvAbortMsg(}\text{rm}\text{)} \overset{\Delta}{=} \quad \land \text{type } \mapsto \text{“Abort”} \odot \text{msgs} \}
\quad \land \text{rmState}' = [\text{rmState EXCEPT } !\text{rm} = \{ \text{“aborted”} \}]
\quad \land \text{UNCHANGED } \langle \text{tmState}, \text{tmPrepared}, \text{msgs} \rangle$

A transition of the distributed system

Next $\overset{\Delta}{=} \quad \lor \text{TMCommit} \lor \text{TMAbort} \
\lor \exists \text{rm} \in \text{RM} : \quad \text{TMRcvPrepared(}\text{rm}\text{)} \lor \text{RMPrep(}\text{rm}\text{)} \lor \text{RMCvCommitMsg(}\text{rm}\text{)}$
\lor \text{RMChoosetAbort(}\text{rm}\text{)} \lor \text{RMCvAbortMsg(}\text{rm}\text{)}$

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

The algorithm is designed to satisfy the following invariant:

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

$\text{TLA}^+$ uses syntax $f[x]$ for function application, e.g., see $\text{rmState}[rm]$. 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 $\text{TLA}^+$. First, there is no fixed order of evaluating the expressions. An operator such as $\text{Next}$ 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, $\text{TLA}^+$ specifications are normally written as a collection of operator definitions. They can be also organized in modules. As the operator $\text{Next}$ describes one step of a system execution, the operators in $\text{TLA}^+$ 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 $\text{let}$-$\text{in}$ expression. We also instantiate modules, in order to obtain a single-module specification, in which the operators $\text{Init}$ and $\text{Next}$ contain only the calls to the built-in $\text{TLA}^+$ operators. The flattening phase is purely syntactic, so we obviously obtain an equivalent $\text{TLA}^+$ specification.

**Note on Recursive Operators.** [Lamport 2018] recently added recursive operators to $\text{TLA}^+$ 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 $\text{TLA}^+$ 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 $\text{TLA}^+$. However, the model checker $\text{TLC}$ interprets expressions $x' = e$ and $x' \in S$ as assignments, if $x'$ has not been assigned a value before. $\text{TLC}$ 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., $\{1, \text{"a"}\}$. 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]:

$$\tau ::= \text{Name} | \text{Bool} | \tau \rightarrow \tau | \text{Set}[\tau] | \text{Seq}[\tau] | \tau \ast \ldots \ast \tau | [n m_1 : \tau, \ldots, n m_k : \tau]$$

The type system rejects some TLA+ expressions that are legal in the untyped language. Importantly, elements of sets with a super type, e.g., $\{1, 2, 3\}$ 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 $\text{false}$ and $\text{true}$ 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., $+: \text{Int} \ast \text{Int} \rightarrow \text{Int}$. Some operators introduce bound variables, e.g., $\exists x \in S : e$ or $\{e : x \in S\}$. As expected, the type of the binding set is computed first, and then the type of $e$ is computed.

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

4 KERA+: THE KERNEL LANGUAGE OF TLA+ EXPRESSIONS

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

A list of KERA+ expressions is given in Table 1. It might seem surprising that very basic operators such as Boolean operators are missing. In fact, they can be expressed with if-then-else:

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

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,... c₁, ..., cₙ (constants) |
| Integers: | i₁ • i₂ where • is one of: +, -, *, ÷, %, ≤, ≥, =, ≠ |
| Sets: | \{e₁, ..., eₙ\} \{x ∈ S: p\} \{e: x ∈ S\} UNION S |
| Control: | ITE(p, e₁, e₂) |
| Quantifiers: | \(\exists x ∈ S: p\) CHOOSE x ∈ S: p FROM e₁, ..., eₙ BY \(θ\) |
| Functions: | \([x ∈ S → e]\) \([e]\) DOMAIN f \([f\ \text{EXCEPT } ![e₁] = e₂]\) |
| Records: | \([nm₁ → e₁, ..., nmₙ → eₙ]\) DOMAIN r e nm |
| Tuples: | \(⟨e₁, ..., eₙ⟩\) \([t[i]\) DOMAIN t |
| Sequences: | \(⟨e₁, ..., eₙ⟩\) \(s[i]\) DOMAIN s \([s\ \text{EXCEPT } ![i] = e]\) |
| | Len(s) \(s \circ t\) Head(s), Tail(s) SubSeq(s, i, j) |

Several KerA* operators do not originate from TLA*:

- **Assignment** \(x’ ∈ S\): Following TLC, under the conditions given by [Kukovec et al. 2018], we treat an expression \(x’ ∈ S\) 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 \(x’ ∈ \{e\}\). We label such assignments with \(x’ ∈ S\), to distinguish them from membership tests \(x’ ∈ S\).

- **Non-deterministic disjunction** \(ϕ₁ ∨ ... ∨ ϕₙ\): This operator formalizes the special form of TLC disjunction. It evaluates to true if and only if the disjunction \(ϕ₁ ∨ ... ∨ ϕₙ\) evaluates to true. However, non-deterministic disjunction adds constraints on the variable assignments: For every \(i, j ∈ 1..n\) and \(i ≠ j\), formula \(ϕ_j\) 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** \(e₁, ..., eₙ\) **BY** \(θ\): This operator returns expression \(e_i\) when \(θ = i\) and \(1 ≤ i ≤ n\); otherwise, it returns an arbitrary value of the same type as \(e₁, ..., eₙ\).

KerA* is a subset of TLA+ — except for the three operators discussed above — and the meaning of the operators coincides with the description in the book by [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 $\tau$: The cells of types Int and Bool are interpreted in SMT as integers and Booleans respectively, whereas the cells of the other types remain uninterpreted. In the following, we use notation $c_i$ or $c_{\text{name}}$ to refer to a cell. We assume fixed a finite set of cells $C$, which contains sufficiently many elements for rewriting a 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: $\{1, 2\} \leadsto \{c_1, 2\} \leadsto \{c_1, c_2\} \leadsto c_3$. We give the precise definition of $\leadsto$ 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 $A = (V, E)$, where $V \subseteq C$ is a finite set, called arena cells, and $E \subseteq V \times (1..|V|) \times V$ is a relation between the cells, called arena edges, that have the following properties:

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

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

- Notation $A, c : \tau$ to introduce the arena $(V', E')$ such that $V' = V(A) \cup \{c\}$ and $E' = E(A)$, provided that $c$ is a fresh cell of type $\tau$, i.e., $c \notin V(A)$.
- Notation $A, c \rightarrow c_1, \ldots, c_n$ to introduce the arena $A'$ such that $V(A') = V(A)$ and $E(A') = E(A) \cup \{(c, i, c_j) | 1 \leq i \leq n\}$.

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 by the cell. For instance, from the edge $c_{15} \xrightarrow{1} c_{11}$ and the tuple type Name $\times$ Int $\times$ $[b : \text{Int}, c : \text{Int}]$, 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.

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

- \(e\): TLA\(^+\) expression
- \(\mathcal{A}\): arena
- \(v\): valuation function
- \(\Phi\): quantifier-free formula

**Example 5.2.** Consider the edge \(e_{41} = (c_4, 1, c_1)\) in Figure 3 (a). If \(en(e_{41})\) evaluates to true, then the cell \(c_1\) belongs to the set encoded by the set \(c_4\); otherwise, \(c_1\) does not belong to the set. \(\blacksquare\)
• $e$ is a KERA* expression, possibly containing cells,
• $\mathcal{A}$ is an arena,
• $\nu$ is a partial function from $\text{Vars}$ to $\mathcal{V}(\mathcal{A})$, which is called binding, and
• $\Phi$ is a set of first-order formulas, which represents SMT constraints.

We define $\rightsquigarrow$ via a set of reduction rules. For instance, the rules (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 $\langle e \mid \mathcal{A} \mid v \mid \Phi \rangle$.

\[
\begin{align*}
\langle b \mid \mathcal{A} \mid v \mid \Phi \rangle & \quad \text{ ($b$ is FALSE or TRUE)} \quad \langle c \mid \mathcal{A}, c : \text{Bool} \mid v \mid \Phi, c = b \rangle & & \text{(BOOL)} \\
\langle c \mid \mathcal{A}, c : \text{Int} \mid v \mid \Phi, c = n \rangle & \quad n \text{ is 0, 1, } -1, \ldots & & \text{(INT)} \\
\langle c_{\ell} \Rightarrow c_r \mid \mathcal{A} \mid v \mid \Phi \rangle & \quad \Rightarrow \text{ is one of } <, \leq, >, \geq, =, \neq & & \text{(INTCMP)} \\
\langle c_{\text{res}} \mid \mathcal{A}, c_{\text{res}} : \text{Bool} \mid v \mid \Phi, c_{\text{res}} \leftrightarrow c_{\ell} \Rightarrow c_r \rangle & & & \text{(INTARITH)}
\end{align*}
\]

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 $\Phi$.

\[
\begin{align*}
\langle c_{\ell} \circ c_r \mid \mathcal{A} \mid v \mid \Phi \rangle & \quad \circ \text{ is one of } +, -, \ast, \div, \% & & \text{(INTARITH)} \\
\langle c_{\text{res}} \mid \mathcal{A}, c_{\text{res}} : \text{Int} \mid v \mid \Phi, c_{\text{res}} = c_{\ell} \circ c_r \rangle & & & \text{(INTARITH)}
\end{align*}
\]

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

\[
\begin{align*}
\langle \text{Op}(e_1, \ldots, e_n) \mid \mathcal{A}_0 \mid v_0 \mid \Phi_0 \rangle & \quad \langle e_i \mid \mathcal{A}_{i-1} \mid v_{i-1} \mid \Phi_{i-1} \rangle & & \text{for } 1 \leq i \leq n \quad \text{(REDARG)} \\
\langle \text{Op}(c_1, \ldots, c_n) \mid \mathcal{A}_n \mid v_n \mid \Phi_n \rangle & & & \text{(REDARG)}
\end{align*}
\]

To apply the reduction system to a KERA* expression $e$, e.g., to $\text{Init}$ and $\text{Next}$, we introduce an initial state $\langle e_0 \mid \mathcal{A}_0 \mid v_0 \mid \Phi_0 \rangle$, whose arena, binding, and SMT constraints are empty. Formally, $\mathcal{A}_0 = (\emptyset, \emptyset)$, $\Phi_0 = \emptyset$, and $v(x) = \bot$ for $x \in \text{Vars}$. 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 $\text{Init}$. Then, we apply the reduction rules until one of the following states is reached: (1) an error state, in which no rule applies, or (2) a terminal state, in which the expression is a single cell. If an error state has been reached, then the expression $e$ is not well-formed.

When a terminal state $c_{\text{term}}$ is reached, and the terminal cell $c_{\text{term}}$ has type Bool, we add the assertion $c_{\text{term}}$ 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, and the terminal sets being the objects of non-set types (or empty sets). Importantly, we only consider finite sets.

Set Enumeration. The simplest way to construct a set is by enumerating its elements, e.g., by writing \{1, 2, 3\}. The rule (ENUM) reduces a set of cells to a fresh cell \(c_{\text{set}}\). The rule links the elements \(c_1, \ldots, c_n\) to \(c_{\text{set}}\) in the arena and adds the constraint \(\text{en}(c_{\text{set}}, i, c_i)\) for each \(1 \leq i \leq n\). Several important observations should be made. First, we only add constraints on the edges from \(c_{\text{set}}\) to the cells \(c_1, \ldots, c_n\), as the reduction rules for sets refer only to the cells pointed by \(c_{\text{set}}\) 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.

\[
\frac{\{c_1, \ldots, c_n\} : \text{Set}[\tau] \mid \mathcal{A} \mid v \mid \Phi}{c_{\text{set}} : \text{Set}[\tau] \mid \mathcal{A}, c_{\text{set}}, c_{\text{set}} \rightarrow c_1, \ldots, c_n \mid v \mid \Phi, \bigwedge_{1 \leq i \leq n} \text{en}(c_{\text{set}}, i, c_i)}
\]

(ENUM)

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

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

\[
\frac{\{x \in c_S : p\} : \text{Set}[\tau] \mid \mathcal{A} \mid v \mid \Phi}{p[c_1/x], \ldots, p[c_n/x] \mid \mathcal{A} \mid \Phi \mid v}
\]

\[
\frac{c_{\text{filter}}^p, \ldots, c_{\text{filter}}^n \mid \mathcal{A}' \mid \Phi' \mid v'}{c_S \rightarrow \mathcal{A} c_1, \ldots, c_n \mid v' \mid \Phi', \text{InFilter}}
\]

(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 \(p[c_1/x]\) for \(1 \leq i \leq n\). (As usual, the notation \(p[e/x]\) means that \(e\) is replaced by \(x\) 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:

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

(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 \(c_S\).

\[
\frac{\text{UNION } c_S : \text{Set}[\tau] \mid \mathcal{A} \mid v \mid \Phi}{c_S \rightarrow \mathcal{A} c_1^1, \ldots, c_n^1 \mid c_S \rightarrow \mathcal{A} c_1^2, \ldots, c_m^2 \mid \text{for } 1 \leq i \leq n
\]

\[
c_U : \text{Set}[\tau] \mid \mathcal{A}, c_U, c_U \rightarrow c_1, \ldots, c_m, c_1^2, \ldots, c_m^2 \mid v \mid \Phi, \text{InUnion}
\]

(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 \(c_i^p\), and the set \(c_S^i\) belongs to \(c_U\):

\[
\text{en}(c_U, idx_{i,j}, c_j^i) \leftrightarrow (\text{en}(c_S^i, j, c_j^i) \land \text{en}(c_S, i, c_S^i)) \text{ for } 1 \leq i \leq n, 1 \leq j \leq m_i,
\]

(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\((c_5, 1, c_2)\) to true and en\((c_4, 2, c_2)\) to false, that is, 2 is a member of the set encoded by \(c_5\) and 2 is not a member of the set encoded by \(c_4\). Equation (InU) produces the following constraints (among others): en\((c_U, 2, c_2)\) \(\leftrightarrow\) en\((c_4, 2, c_2)\) \& en\((c_6, 1, c_4)\) and en\((c_U, 3, c_2)\) \(\leftrightarrow\) en\((c_5, 1, c_2)\) \& en\((c_6, 2, c_5)\). As a result, en\((c_U, 2, c_2)\)
is false, whereas $en(c_{U}, 3, c_{2})$ is true. There is no contradiction here, as for the set membership of $c_{2}$ in $c_{U}$, it is sufficient to find one enabled edge, that is, $(c_{U}, 3, c_{2})$.

**Set Map.** By definition, $\{ e : x \in S \}$ constructs the set $T$ with the following property: For every $z$, 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.

![Diagram showing rules for set comprehension](image)

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.

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 $c_{S}$:

$$en(c_{T}, i, c_{i}^{e}) \leftrightarrow en(c_{S}, i, c_{i}) \text{ for } 1 \leq i \leq n$$

*(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 $en(c_{S}, 3, c_{4})$ to true and $en(c_{S}, 2, c_{3})$ to false. Hence, $en(c_{T}, 3, c_{1})$ holds true and $en(c_{T}, 2, c_{1})$ does not. Still, $c_{1}$ belongs to the set encoded by $c_{T}$, as the edge $(c_{T}, 3, c_{1})$ is enabled.

**Integer Interval $a..b$.** This operator is quite often used in TLA$^*$ to define the set $\{ i \in \mathbb{Z} : a \leq i \leq b \}$.

The latter set cannot be defined in 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 $S \ni a..b$ that can be filtered by the KERA$^*$ expression $\{ i \in S : a \leq i \land i \leq b \}$. 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, the (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 $c_{S} = c_{T}$.

$$\langle c_{S} = c_{T} | \mathcal{A} | v | \Phi \rangle \Rightarrow \langle c_{eq} : \text{Bool} | \mathcal{A}' | v' | \Phi' \rangle$$

*(SetEq)*

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

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

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

$$\text{notSeen}_i = \bigwedge_{1 \leq j < i} (en(c_S, j, c_j) \rightarrow c_j \neq c_i) \text{ for } 0 < i \leq n \quad (2)$$

Hence, $\text{Cardinality}(c_S) = k_n$. 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 $\text{FROM } e_1, \ldots, e_n \text{ BY } \theta$, where $\theta$ is an integer constant and $e_1, \ldots, e_n$ are TLA+ expressions of the same type $\tau$. The meaning of this operator is as follows: If $\theta \in 1..n$, then $\text{FROM } e_1, \ldots, e_n \text{ BY } \theta$ returns $e_{\theta}$; Otherwise, it returns an arbitrary value of type $\tau$. The constant $\theta$ defines the value to be picked from the sequence $e_1, \ldots, e_n$. Hence, we call it an oracle.

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

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

$$\langle \text{FROM } c_1, \ldots, c_n \text{ BY } \theta \mid \mathcal{A} \mid v \mid \Phi \rangle \quad c_1 : \tau_1, \ldots, c_n : \tau \quad \tau \text{ is basic}$$

$$\langle c_{\text{pick}} \mid \mathcal{A}, c_{\text{pick}} : \tau \mid v \mid \Phi, \bigwedge_{1 \leq i \leq n} (\theta = i \rightarrow c_{\text{pick}} = c_i) \rangle \quad \text{(FROMBASIC)}$$
Fig. 6. (a) The arena constructed for the tuple \( (1, \text{true}, "abc") \), 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 \([a \mapsto \text{Int}, b \mapsto \text{Bool}, c \mapsto \text{Name}]\), assuming that the expressions 1, true, and "abc" were rewritten into cells \( c_4 \), \( c_5 \), and \( c_6 \).

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

The rule (\textsc{FromSet}) works in two steps. First, for every index \( j \in 1..m \), it picks an element \( c_{\text{pick}}^j \) among the \( j \)th elements of the sets \( c_1, \ldots, c_n \). Importantly, the operators \( \text{FROM} c_i \), \( \ldots, c_n \) \( \text{BY} \ \theta \) are using the same oracle \( \theta \) for every \( j \in 1..m \). As a result, they pick the respective elements from the same set \( c_\theta \). Second, the resulting set \( c_{\text{res}} \) points to the picked elements \( c_{\text{pick}}^1, \ldots, c_{\text{pick}}^m \).

\[
\begin{align*}
&\langle \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \mid A_0 \mid v_0 \mid \Phi_0 \rangle, \\
&\quad \langle c_i : \text{Set}[\tau] \text{ for } 1 \leq i \leq n \text{ and } \theta : \text{Int} \rangle, \\
&\quad \langle \text{FROM} c_i^1, \ldots, c_i^n \text{ BY } \theta \mid A_{j-1} \mid v_{j-1} \mid \Phi_{j-1} \rangle, \\
&\quad c_i \rightarrow A_i \ c_i^1, \ldots, c_i^m \text{ for } 1 \leq i \leq n, \\
&\quad \langle c_{\text{res}} \mid A, c_{\text{res}} : \text{Set}[\tau], c_{\text{res}} \rightarrow c_{\text{pick}}^1, \ldots, c_{\text{pick}}^m \mid v_m \mid \Phi_m, \text{InPicked} \rangle \quad \text{(FROMSet)} \\
\end{align*}
\]

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

\[
en(c_{\text{res}}, j, c_{\text{pick}}^j) \leftrightarrow \bigvee_{1 \leq i \leq n} \theta = i \land en(c_i, j, c_i^j) \text{ for } 1 \leq j \leq m \quad \text{ (InPicked)}
\]

**Example 7.1.** Figure 5 shows an example of the rule applied to \textsc{FROM} \( c_1, c_2 \) \text{ BY } \theta. The cells \( c_1 \) and \( c_2 \) have type \( \text{Set}[\tau] \), each of them pointing to three element cells \( c^1_1, c^2_1, \) and \( c^3_1 \) for \( i \in \{1, 2\} \). The rule first applies \textsc{FROM} \( c^1_j, c^2_j \) \text{ BY } \theta \) three times for \( j \in \{1, 2, 3\} \) to pick one element \( c_{\text{pick}}^j \) from each pair. Note that use of \( \theta \) guarantees us that the elements are drawn from the same set. The resulting cell \( c_{\text{res}} \) is pointing to the three picked cells \( c_{\text{pick}}^1, c_{\text{pick}}^2, \) and \( c_{\text{pick}}^3 \).

**Picking Other Values.** We have also defined the rules for picking a value from: a set of functions, a set of tuples, a set of records, a set of sequences, and a powerset (constructed with \textsc{Subset} \( S \)). They are similar to (\textsc{FromBasic}) and (\textsc{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., \texttt{tuple[3]} or \texttt{record.name}, but not \texttt{tuple[x]} and \texttt{record[x]}, where \(x\) is a variable. This is usually the case for TLA\(^+\) specifications.

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

\[
\begin{align*}
\langle c_1, \ldots, c_n \rangle & : \tau_1 \times \cdots \times \tau_n \mid \mathcal{A} \mid v \mid \Phi \\
\langle c_{\text{new}} \mid \mathcal{A}, c_{\text{new}} : \tau_1 \times \cdots \times \tau_n, c_{\text{new}} \rightarrow c_1, \ldots, c_n \mid v \mid \Phi \rangle & \quad \text{(TupCtor)}
\end{align*}
\]

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

\[
\begin{align*}
\langle c_t[i] \mid \mathcal{A} \mid v \mid \Phi \rangle & \quad c_t \rightarrow_{\mathcal{A}} c_1, \ldots, c_n \quad i \in \{1, \ldots, n\} \\
\langle c_i \mid \mathcal{A} \mid v \mid \Phi \rangle & \quad \text{(TupApp)}
\end{align*}
\]

**Tuple Domain.** For a tuple \(t\) of type \(\tau_1 \times \cdots \times \tau_n\), the expression \textsc{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 \textsc{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 \(\{(x, f[x]) : x \in \text{domain} f\}\). 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 \textsf{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 \(\{(x, e) : x \in S\}\). This rule produces a cell \(c_{\text{rel}}\) that encodes the associated relation \(c_{\text{rel}}\) of type \(\text{Set}[\tau_1 \times \tau_2]\), where \(\tau_1\) is the type of elements of \(S\), and \(\tau_2\) is the type of \(e\). We add a cell \(c_f\) of type \(\tau_1 \rightarrow \tau_2\) and make it point to \(c_{\text{rel}}\), that is, \(c_f \rightarrow_{\mathcal{A}} c_{\text{rel}}\). The rule (FuncTor) produces \(c_f\) as a result.

**Function Domain (FuncDom).** Assuming that \(f\) is reduced to a cell \(c_f\), we rewrite \textsc{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 (FuncExc).** In TLA\(^+\), an expression \(f\) \texttt{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 \text{funrel}(c_f)\) as follows:

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

(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.

\[
\begin{align*}
\frac{c_{\text{fun}}[c_{\text{arg}}] \mid \mathcal{A} \mid v \mid \Phi}{\langle c_{\text{fun}}[c_{\text{arg}}] \mid \mathcal{A} \mid v \mid \Phi \rangle} & \quad \text{FunApp} \\
\frac{\langle \text{from } c_1, \ldots, c_n \text{ by } c_{\text{ora}} \mid \mathcal{A}, c_{\text{ora}} : \text{Int} \mid v \mid \Phi, 0 \leq c_{\text{ora}} \leq n \rangle}{\langle c_{\text{pair}} \mid \mathcal{A}_2 \mid \Phi_2 \mid v_2 \rangle} & \quad \text{(FunApp)}
\end{align*}
\]

First, the rule (FunApp) introduces an integer oracle \( c_{\text{ora}} \), which points either to a cell from \( c_1, \ldots, c_n \) (when \( 1 \leq c_{\text{ora}} \leq n \)), or an arbitrary cell of proper type (when \( c_{\text{ora}} = 0 \)). Second, a cell \( c_{\text{pair}} \) is picked using the operator \( \text{from } c_1, \ldots, c_n \text{ by } c_{\text{ora}} \). This is the tuple that comprises a function argument and the respective result, so the rule (FunApp) returns the function result \( c_{\text{pair}}[2] \). 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 \( c_{\text{arg}} \). Importantly, as the rule uses equality, we require that the lazy equality constraints \( c_{\text{arg}} = c_i[1] \) are generated for \( 1 \leq i \leq n \).

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

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

10 CONTROL OPERATORS AND QUANTIFIERS

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

\[
\begin{align*}
\langle \text{ITE}(c_p, c_1, c_2) : \tau \mid \mathcal{A} \mid v \mid \Phi \rangle & \quad \text{ITE} \\
\frac{\langle \text{from } c_1, c_2 \text{ by } \theta \mid \mathcal{A}, \theta : \text{Int} \mid v \mid \Phi, 1 \leq \theta \leq 2 \rangle}{\langle c_{\text{res}} \mid \mathcal{A}_2 \mid v_2 \mid \Phi_2 \rangle} & \quad \text{ITE}
\end{align*}
\]

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

Assignments. An assignment \( x' \in C_S \) in KERA\(^+\) specifies that a variable \( x' \) takes a value from the set \( S \). Since any element of the set may be chosen, we use \( \text{from } c_1, \ldots, c_n \text{ by } \theta \) 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 \(x'\).

\[
\langle x' \in c_S \mid A \mid v \mid \Phi \rangle \quad c_S \rightarrow_A c_1, \ldots, c_n
\]

\[
\langle \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \mid A, \theta : \text{Int} \mid v \mid \Phi, 0 \leq \theta \leq n \rangle
\]

\[
\langle c \mid A_2 \mid v_2 \mid \Phi_2 \rangle
\]

\[
\langle \text{TRUE} \mid A_2 \mid v_2[x \mapsto c] \mid \Phi_2, \theta = 0 \leftrightarrow \bigwedge_{1 \leq i \leq n} \neg en(c_S, i, c_i), \bigwedge_{1 \leq i \leq n} (\theta \neq i \lor en(c_S, i, c_i)) \rangle \quad \text{(ASGN)}
\]

We omit the rules for the assignments \(f' \in \text{SUBSET} S\) and \(f' \in [S \rightarrow T]\) for brevity.

**Substitution.** A variable \(x\) can be replaced with the cell given by a valuation \(v\):

\[
\langle x \mid A \mid v \mid \Phi \rangle \quad x \in \text{Vars}
\]

\[
\langle v(x) \mid A \mid v \mid \Phi \rangle \quad \text{(Sub)}
\]

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

\[
\langle \exists x \in c_S : p \mid A \mid v \mid \Phi \rangle \quad c_S \rightarrow_A c_1, \ldots, c_n
\]

\[
\langle p[c_1/x] \oplus \ldots \oplus p[c_n/x] \mid A \mid v \mid \Phi \rangle \quad \text{(EXISTS)}
\]

Replacing an existential quantifier with a disjunction may seem to be suboptimal. However, we cannot avoid it, as existential quantification may be used to express universal quantification, e.g., \(\neg \exists x \in c_S : p\) is located under a universal quantifier. If this is not the case, we introduce a Skolem constant \(c \in c_S\) and produce the expression \(p[c/x]\) instead of the disjunction. As expected, this optimization significantly reduces the number of SMT constraints.

**Operator choose.** By definition, \(\text{choose} \ x \in S : p\) 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, \(\text{choose}\) is deterministic: Two expressions \(\text{choose} \ x \in S : p\) and \(\text{choose} \ y \in T : q\) have equal values, if the filtered sets are equal, that is, \(\{x \in S : p\} = \{y \in T : q\}\).

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

\[
\langle \text{choose} \ x \in S : p \mid A \mid v \mid \Phi \rangle
\]

\[
\langle \{x \in S : p\} \mid A \mid v \mid \Phi \rangle
\]

\[
\langle c_F : \tau \mid A_2 \mid v_2 \mid \Phi_2 \rangle
\]

\[
\langle \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \mid A_2, \theta : \text{Int} \mid v_2 \mid \Phi_2, 0 \leq \theta \leq n \rangle
\]

\[
\langle c_{\text{res}} \mid A_3 \mid v_3 \mid \Phi_3 \rangle
\]

\[
\text{(Choose)}
\]

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

$$
\begin{align*}
\langle e_1 \oplus e_2 \mid A_0 \mid v_0 \mid \Phi_0 \rangle & \quad \langle e_i \mid A_{i-1} \mid v_0 \mid \Phi_{i-1} \rangle \quad i = 1, 2 \\
\text{(FROM } v_1(x'), v_2(x') \text{ BY } \theta \mid A_2, \theta : \text{Int} \mid v_0 \mid \Phi_2, \theta \in \{1, 2\}) & \quad \langle c_1 \mid A_1 \mid v_1 \mid \Phi_1 \rangle \\
\langle c_x \mid A_3 \mid v_0 \mid \Phi_3 \rangle & \quad \langle c_r \mid A_3, c_r : \text{Bool} \mid v_0 \circ [x' \mapsto c_x] \mid \Phi, c_r \mapsto c_1 \lor c_2, \theta = 1 \rightarrow c_1, \theta = 2 \rightarrow c_2 \rangle \quad \text{(NDC)}
\end{align*}
$$

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 $M = \langle D, I \rangle$:

1. $D$ is a domain. It is a disjoint union of sets $D_1, \ldots, D_n$, each $D_i$ contains values of type $\tau_i$.
2. $I$ is an interpretation. It assigns values from the domain to the constants and Kera\(^{+}\) operators.

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

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

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

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

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

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

THEOREM 11.2. Every sequence of ARS reductions $s_0 \sim s_1 \sim \ldots$ 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 \( s_0 \leadsto \ldots \leadsto s_m \) be a sequence of states produced by an abstract reduction system, and \( s_i = \langle e_i \mid \mathcal{A}_i \mid v_i \mid \Phi_i \rangle \) for \( 1 \leq i \leq m \). Assume that \( e_0 \) is a formula, that is, it has type \( \text{Bool} \). The formula \( e_0 \) is satisfiable if and only if the constraint \( e_m \land \Phi_m \) is satisfiable.

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

In the following, we give the idea of our proof of Theorem 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 \( s_0 \leadsto \ldots \leadsto s_m \) from an initial state \( s_0 \) leading to \( s_m \). Similarly, a transition is reachable, if it originates from a reachable state.

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

**Invariants of the Reduction System.** In order to prove soundness of the translation to SMT, we formulate six invariants on the reachable states and transitions of the abstract reduction system. Proposition 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 \( \langle e \mid \mathcal{A} \mid v \mid \Phi \rangle \) 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 \( \Phi \):

**Invariant 2.** In every reachable state \( \langle e \mid \mathcal{A} \mid v \mid \Phi \rangle \) of the ARS, the following holds:
1. Every cell \( c \) appears in either the expression \( e \) or the formula \( \Phi \) if and only if it appears in \( \mathcal{A} \).
2. Arena \( \mathcal{A} \) has an edge \( (c_{\text{set}}, i, c_{\text{elem}}) \) iff the formula \( \Phi \) contains the constant \( e_n(c_{\text{set}}, i, c_{\text{elem}}) \).

Invariant 3 ensures that the reduction rules produce suitable models:

**Invariant 3.** Let \( s_{\text{before}} \leadsto s_{\text{after}} \) be a reachable transition in the ARS, and \( M_{\text{before}} \) a suitable model for \( s_{\text{before}} \). An extended structure \( M_{\text{after}} \) from \( M_{\text{before}} \) is also suitable for \( s_{\text{after}} \).

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

**Invariant 4.** Let \( \langle e \mid \mathcal{A} \mid v \mid \Phi \rangle \) be a reachable state of the ARS, and \( M \) be its extended model. Assume that \( c_{\text{set}} \) is a set cell in the arena \( \mathcal{A} \). Then, the following holds:
1. Assume that \( c_{\text{set}} \rightarrow \mathcal{A} c_1, \ldots, c_n \), for some \( n \geq 0 \), and \( c_{\text{set}} \) is introduced by a rule different from (FROM\( \mathcal{A} \)). Then, the following holds: \( \llbracket c_{\text{set}} \rrbracket^M \subseteq \{ \llbracket c_1 \rrbracket^M, \ldots, \llbracket c_n \rrbracket^M \} \).
2. Assume that \( c_{\text{set}} \) is a reduction of the expression \( \text{FROM} c_1, \ldots, c_n \text{ BY } \theta \) with \( 1 \leq \llbracket \theta \rrbracket^M \leq n \) and \( c_{\text{set}} \rightarrow \mathcal{A} c_{\text{pick}}^1 \ldots c_{\text{pick}}^n \). Then, the following holds: \( \llbracket c_{\text{set}} \rrbracket^M \subseteq \{ \llbracket c_{\text{pick}}^1 \rrbracket^M, \ldots, \llbracket c_{\text{pick}}^n \rrbracket^M \} \).

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

**Invariant 5.** Let \( \langle e \mid \mathcal{A} \mid v \mid \Phi \rangle \) 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 \( c_{\text{rel}} \) of type \( \text{Set}[\tau_1 \times \tau_2] \) such that the function cell \( c_f \) is pointing to it: \( c_f \rightarrow \mathcal{A} c_{\text{rel}} \).
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 \( c_{rel}^f \).

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

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

**Proposition 11.5.** Let \( s_0 \leadsto \ldots \leadsto s_m \) be a sequence of states produced by an abstract reduction system. Then, Invariant 3 is preserved by every transition \( s_i \leadsto s_{i+1} \) 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 \( \land \) 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 \( \langle \text{Inv} \land T_i \land \neg \text{Inv}' \mid A_0 \mid v_0 \mid \Phi_0 \rangle \) and obtains the final state \( \langle c_i^{\text{final}} \mid A_k \mid v_k \mid \Phi_k \rangle \). The tool asks the solver, whether \( \Phi_k \land c_i^{\text{final}} \) is satisfiable. If this is the case, the tool reports a counterexample to induction, which is obtained from the SMT model. If this is not the case for all \( 1 \leq i \leq m \), the inductive invariant holds true.

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

**Bounded Model Checking.** Given a safety property \( P \) and a number \( k \geq 0 \), this technique verifies, whether there is a computation of length up to \( k \) that violates the property \( P \) in one of the computation states. Equations (3)–(4) show a series of reductions that are used to encode an computation of length \( k \). The values of the variables \( x' \) computed at step \( i \) are used as the values of the variables \( x \) at step \( i + 1 \). This is done by changing the variable substitution \( v_i \) to \( v_i[x \mapsto x', x' \mapsto \bot] \).

\[
\langle \text{Init'} \mid A_0 \mid v_0 \mid \Phi_0 \rangle \leadsto^* \langle c_1 \mid A_1 \mid v_1 \mid \Phi_1 \rangle \tag{3}
\]

\[
\langle \text{Next} \mid A_i \mid v_i[x \mapsto x', x' \mapsto \bot] \mid \Phi_i \rangle \leadsto^* \langle c_{i+1} \mid A_{i+1} \mid v_{i+1} \mid \Phi_{i+1} \rangle \text{ for } 1 \leq i \leq k \tag{4}
\]

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 \( \Phi_i \land \neg c_i^P \land \bigwedge_{1 \leq j \leq i} c_j \) states that the property \( P \) is violated after the transition \( i - 1 \). Satisfiability of this formula gives us a counterexample.

\[
\langle \neg P \mid A_i \mid v_i \mid \Phi_i \rangle \leadsto^* \langle c_i^P \mid A_i^P \mid v_i^P \mid \Phi_i^P \rangle \text{ for } 1 \leq i \leq k \tag{5}
\]
Table 2. The list of TLA+ benchmarks

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

### 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 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 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
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.

<table>
<thead>
<tr>
<th>#</th>
<th>Name</th>
<th>APALACHE</th>
<th>TLC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>memory</td>
</tr>
<tr>
<td>1</td>
<td>Traffic</td>
<td>6s</td>
<td>221M</td>
</tr>
<tr>
<td>2</td>
<td>Prisoners-4</td>
<td>3m19s</td>
<td>355M</td>
</tr>
<tr>
<td>3</td>
<td>Bakery-5</td>
<td>18ms</td>
<td>774M</td>
</tr>
<tr>
<td>4</td>
<td>EWD840-4</td>
<td>56s</td>
<td>1.13G</td>
</tr>
<tr>
<td>5</td>
<td>EWD840-10</td>
<td>13m</td>
<td>1.17G</td>
</tr>
<tr>
<td>6</td>
<td>SimpAlloc-2-2</td>
<td>34s</td>
<td>371M</td>
</tr>
<tr>
<td>7</td>
<td>SimpAlloc-5-3</td>
<td>2h56m</td>
<td>722M</td>
</tr>
<tr>
<td>8</td>
<td>bcastFolk-4</td>
<td>20s</td>
<td>712M</td>
</tr>
<tr>
<td>9</td>
<td>bcastFolk-20</td>
<td>1m69s</td>
<td>1.11G</td>
</tr>
<tr>
<td>10</td>
<td>bcastByz-4</td>
<td>9m14s</td>
<td>1.13G</td>
</tr>
<tr>
<td>11</td>
<td>bcastByz-6</td>
<td>3h00m</td>
<td>1.18G</td>
</tr>
<tr>
<td>12</td>
<td>TwoPhase-3</td>
<td>1m13s</td>
<td>475M</td>
</tr>
<tr>
<td>13</td>
<td>TwoPhase-7</td>
<td>44m</td>
<td>516M</td>
</tr>
<tr>
<td>14</td>
<td>Paxos-3</td>
<td>1h37m</td>
<td>825M</td>
</tr>
<tr>
<td>15</td>
<td>Paxos-5</td>
<td>7h09m</td>
<td>1015M</td>
</tr>
<tr>
<td>16</td>
<td>Raft-5</td>
<td>2h47m</td>
<td>1.18G</td>
</tr>
</tbody>
</table>

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.

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

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 i-th 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:

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

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

[Swamy et al. 2016] designed the general-purpose functional programming language F\textsuperscript{*} with effects aimed at program verification. Like Boogie, this language utilizes SMT solvers as backend 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].

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.
REFERENCES


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


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


