

Symbolic Model Checking for TLA⁺ Made Faster

Rodrigo Otoni¹, Igor Konnov², Jure Kukovec², Patrick Eugster¹, and
Natasha Sharygina¹

¹ Università della Svizzera italiana, Lugano, Switzerland
`otonir@usi.ch, eugstp@usi.ch, sharygin@usi.ch`

² Informal Systems, Vienna, Austria
`igor@informal.systems, jure@informal.systems`

In the proceedings of TACAS'23: https://doi.org/10.1007/978-3-031-30823-9_7

Abstract. The need to provide formal guarantees about the behaviour of the algorithms underpinning modern distributed systems became evident in recent years. This interest made apparent the complexities involved in applying verification techniques in a distributed setting, with significant effort being made in both academia and industry to aid in this endeavour. Many formalisms have been proposed to tackle the difficulties faced by practitioners, with one that has seen widespread use in industry being TLA⁺, adopted, for instance, by Amazon Web Services. TLA⁺ provides engineers with a way of specifying both systems and desired properties, and is supported by a number of verification tools. Despite their extensive use, such tools suffer considerably from lack of scalability. To solve this, we propose a novel encoding of TLA⁺ into SMT constraints to improve symbolic model checking efficiency. Our insight is the need to provide the SMT solver with structural information about the TLA⁺ specification encoded, i.e., how data structures and their component elements interact, which we do by relying on the SMT theory of arrays. We implemented our approach by modifying the SMT-based model checker APALACHE and evaluated it against comparable tools. Our results show that our approach outperforms existing ones on a number of benchmarks, with an order of magnitude improvement in checking time.

Keywords: Model checking · SMT arrays · Distributed algorithms

1 Introduction

Distributed systems are ubiquitous in the modern world, with many companies directly relying on them to conduct business. Due to this, the ability to ensure that a distributed system is operating correctly is paramount. The search for correctness guarantees led to an influx of interested parties adopting formal verification methodologies in recent years. One of the most famous example of this trend is probably the adoption of TLA⁺ [17] by Amazon Web Services [19]. TLA⁺ is a specification language based on the temporal logic of actions (TLA) which allows users to describe the expected behaviour of a system, while abstracting away implementation details that do not impact high-level properties, e.g., memory management. With TLA⁺ specifications at hand, Amazon engineers rely on model checking for correctness guarantees of systems such as DynamoDB [23].

Despite recent interest and advances, the verification of distributed systems remains notoriously difficult. This is mainly due to the fact that, given their distributed nature, distributed algorithms’ executions admit numerous potential interleavings of steps, with state-spaces generally growing exponentially with the number of participants. In the case of TLA^+ , a handful of tools are available to aid in verification [14]. TLC [27] is an explicit-state model checker that enumerates all reachable states of the given system. APALACHE [13] is a symbolic bounded model checker that uses a satisfiability modulo theories (SMT) encoding of states in order to better tackle the state-space explosion problem. TLAPS [6] is an interactive proof system that enables the proving of properties without the need of exploring the state-space itself. Despite providing the benefit of verifying specifications with infinite state-spaces, and efforts being made towards partial automation [18], TLAPS adoption is still slow, with engineers favouring the push-button automation provided by model checkers.

In this work we focus on symbolic model checking for TLA^+ , as spearheaded by the SMT encoding which underpins APALACHE, but provide insights into SMT-based model checking that may generalise to other contexts. The encoding of TLA^+ into SMT done by APALACHE removes all structural information present in the encoded specification, with all TLA^+ data structures being represented via uninterpreted constants in the generated SMT formula. The information not forwarded to the SMT solver has the potential to significantly improve solving efficiency. We propose an alternative SMT encoding that makes full use of the SMT theory of arrays [8] to encoded the main TLA^+ data structures, i.e., sets and functions, with the goal of improving solving performance, which is the determining factor in overall model checking performance.

Concretely, we modify APALACHE’s abstract reduction system (ARS) to generate constraints in the SMT theory of arrays, while relying on its preprocessing infrastructure, as shown in Figure 1. APALACHE rewrites the input specification into the KerA^+ verification-friendly fragment of TLA^+ [13] and then applies ARS rules to generate the SMT formula to be solved. We implemented our encoding in APALACHE and compared it with APALACHE’s constants encoding and TLC. Our experiments indicate that embedding structural information into the SMT formulas has a significant impact on performance. Our contributions are:

1. Formalisation of a TLA^+ encoding into the SMT theory of arrays;
2. Development of a robust open-source implementation of our encoding;
3. Evaluation via checking agreement on three asynchronous protocols.

The paper is structured as follows: background is given in Section 2, the arrays-based encoding and its evaluation are presented in Sections 3 and 4, related work is discussed in Section 5, and our final remarks are made in Section 6.

2 Background

In this section we introduce the basics of TLA^+ , its KerA^+ fragment used to represent TLA^+ ’s core, the approach to generate SMT constraints from KerA^+ via abstract reduction, and finally the SMT theory of arrays.

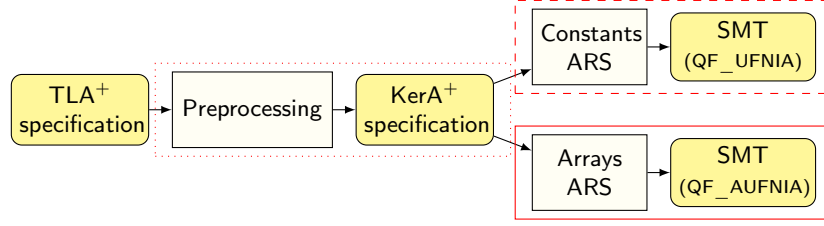


Fig. 1: Overview of the symbolic model checking for TLA⁺. The dotted box highlights the identification of symbolic transitions from [16] and the rewriting into KerA⁺. The dashed box highlights the encoding based on uninterpreted constants from [13]. The solid box highlights the arrays-based encoding we propose.

2.1 TLA⁺

We introduce TLA⁺ via a specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [5], shown in Figure 2. Here we focus on the most relevant TLA⁺ constructs, with further details being available in [17].

The first notable aspect of TLA⁺ is that specifications may be parametrised, e.g., the number of processes and faults may not be fixed. In our example, the keyword `CONSTANTS`, in line 3, is used to declare its parameters: N , the total number of processes, and T and F , the maximal and actual number of faulty processes. It is important to understand, however, that while a specification may be parametrised, model checking can only be carried out for a specific instance of the protocol at a time, e.g., $N = 4$ and $T = F = 1$. Parameter declarations are followed by variable declarations, by the use of the `VARIABLES` keyword, in line 4. Variables define the states of the state-machine that the specification describes, with each state being defined by the combination of the values held by each variable. In our example, each state is defined by the values of *sentEcho*, *sentReady*, *rcvdEcho*, *rcvdReady*, and *pc*.

The remaining TLA⁺ operators describe state-machine transitions or properties to be checked, and are defined using $\triangleq$. Two operators are of special significance, one that defines the initial-state predicate and one that plays the role of the transition operator. In our example, these operators are *Init*, in line 8, and *Next*, in line 22. Concretely, *Init* defines the starting point for state-space exploration and *Next* defines the exploration itself. Transitions are guided by constraints that must hold in both pre-transition states, represented by non-primed variables, and post-transition states, represented by primed variables.

Specifications may optionally define invariants, i.e., properties that should hold in every reachable state. There is no special syntax for invariants, and they are provided by name to model checkers at invocation time. In our example, we have one invariant, *NoDecide*, in line 26. A specification satisfies *NoDecide* if no state reachable from *Init* via any number of *Next* transitions has $pc[p] = \text{“AC”}$, for some $p \in \text{Corr}$. Abstractly, this invariant holds iff *Decide* can never be taken.

```

1  ┌────────────────────────────────── MODULE ABA ───────────────────────────────────┐
2  EXTENDS Integers, FiniteSets
3  CONSTANTS N, T, F
4  VARIABLES sentEcho, sentReady, rcvdEcho, rcvdReady, pc
5  Corr  $\triangleq$   $1 \dots (N - F)$  The set of correct processes
6  Byz  $\triangleq$   $(N - F + 1) \dots N$  The set of Byzantine processes
7  Proc  $\triangleq$   $1 \dots N$  The set of all processes
8  Init  $\triangleq$   $\wedge pc \in [Corr \rightarrow \{ "V0", "V1" \}]$ 
9            $\wedge rcvdEcho = [p \in Corr \mapsto \{ \}] \wedge rcvdReady = [p \in Corr \mapsto \{ \}]$ 
10           $\wedge sentEcho \in \text{SUBSET } Byz \quad \wedge sentReady \in \text{SUBSET } Byz$ 
11  Receive(p, nextEcho, nextReady)  $\triangleq$  ... Omitted for brevity
12  SendEcho(p, nextEcho, nextReady)  $\triangleq$  ... Omitted for brevity
13  SendReady(p, nextEcho, nextReady)  $\triangleq$ 
14       $\wedge pc[p] = \text{"EC"}$ 
15       $\wedge \vee \text{Cardinality}(\text{nextEcho}) \geq (N + T + 2) \div 2$ 
16       $\vee \text{Cardinality}(\text{nextReady}) \geq T + 1$ 
17       $\wedge pc' = [pc \text{ EXCEPT } !p = \text{"RD"}] \wedge sentReady' = sentReady \cup \{p\}$ 
18       $\wedge \text{UNCHANGED } sentEcho$ 
19  Decide(p, nextReady)  $\triangleq$ 
20       $\wedge pc[p] = \text{"RD"} \wedge \text{Cardinality}(\text{nextReady}) \geq 2 * T + 1$ 
21       $\wedge pc' = [pc \text{ EXCEPT } !p = \text{"AC"}] \wedge \text{UNCHANGED } \langle sentEcho, sentReady \rangle$ 
22  Next  $\triangleq$   $\exists p \in Corr, nextEcho \in \text{SUBSET } sentEcho, nextReady \in \text{SUBSET } sentReady :$ 
23       $\wedge \text{Receive}(p, nextEcho, nextReady)$ 
24       $\wedge \vee \text{SendEcho}(p, nextEcho, nextReady) \vee \text{SendReady}(p, nextEcho, nextReady)$ 
25       $\vee \text{Decide}(p, nextReady) \vee \text{UNCHANGED } \langle pc, sentEcho, sentReady \rangle$ 
26  NoDecide  $\triangleq$   $\forall p \in Corr : pc[p] \neq \text{"AC"}$  Invariant stating that processes never Decide
27 └──────────────────────────────────────────────────────────────────────────────────┘

```

Fig. 2: Example of a TLA^+ specification, based on the asynchronous Byzantine agreement protocol by Bracha and Toueg [5]; simplifications made for brevity.

2.2 KerA⁺

TLA^+ provides users with a myriad of ways of specifying systems. This richness, although being one its strengths, adds significant difficulty to the generation of SMT constraints. To overcome this challenge, TLA^+ specifications are rewritten into a more compact language, KerA⁺, before being checked. From KerA⁺, the ARS can generate SMT constraints in a simpler and provably sound way.

The KerA⁺ language consists of a small subset of TLA^+ conjoined with four additional constructs not originating from TLA^+ , and is able to express almost all TLA^+ expressions. It contains constructs for the manipulation of sets, functions, records, tuples, and sequences, as well as integer arithmetic operators, Boolean and integer literals, and constants, with all data structures having a bounded size. The semantics of KerA⁺ derive directly from the TLA^+ constructs it uses, with the non- TLA^+ based constructs, which help simplify the rewriting system, having simple control semantics. The correctness of the rewriting itself is guaranteed by construction. One example is the rewriting of $S \cup T$ into the set comprehension $\{x \in S : x \in T\}$. Further KerA⁺ details are available in [13].

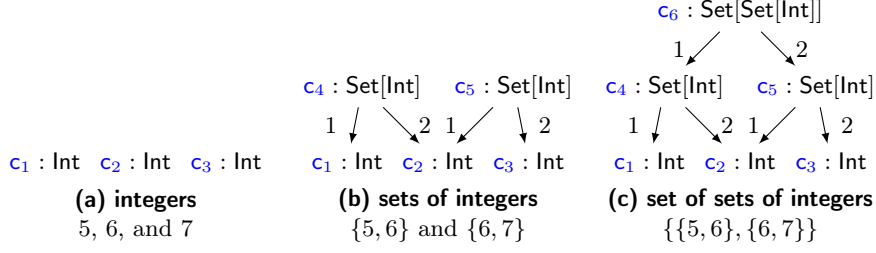


Fig. 3: Illustration of three arenas. The captions describe the modelled elements with the overapproximation $c_1 = 5$, $c_2 = 6$, $c_3 = 7$, $c_4 = \{5, 6\}$, $c_5 = \{6, 7\}$, and $c_6 = \{\{5, 6\}, \{6, 7\}\}$. Note that the concrete value of a cell can be given by any of the possible subtrees having said cell as a root, e.g., for c_6 we have that $\exists c_4 \in \mathcal{P}(\{5, 6\}), c_5 \in \mathcal{P}(\{6, 7\}) . c_6 \in \mathcal{P}(\{c_4, c_5\})$; $\mathcal{P}$ stands for power set.

2.3 Abstract Reduction System

In order to verify a specification in KerA^+ we generate a SMT formula that is equisatisfiable to it. To do so, we use an abstract reduction system (ARS) which iteratively applies reduction rules that transform KerA^+ expressions into SMT constraints. The core of the ARS is the *arena*, a graph structure that overapproximates the specification's data structures and guides rule application. The rules collapse KerA^+ expressions into *cells*, which represent the symbolic evaluation of these expressions, with the cells then being used as vertices in the arena. The arena edges represent the data structures overapproximation, e.g., a cell representing a set will have directed edges to the cells representing all its potential elements, as illustrated in Figure 3. The reduction process terminates when the initial KerA^+ expression e is collapsed into a single cell c , producing a SMT formula Φ in the process, such that $c \wedge \Phi$ is equisatisfiable to e ; equisatisfiability relies on the boundedness of the data structures and is detailed in Section 3.3. The satisfiability of e can then be checked by forwarding $c \wedge \Phi$ to a SMT solver.

Formally, the ARS is defined as $(\mathcal{S}, \rightsquigarrow)$, with $\mathcal{S}$ being the set of ARS states and $\rightsquigarrow \subseteq \mathcal{S} \times \mathcal{S}$ being the transition relation. A state $(e, \mathcal{A}, \nu, \Phi) \in \mathcal{S}$ is a four-tuple containing a KerA^+ expression e , an arena $\mathcal{A}$, a binding of names to cells ν , and a first-order formula Φ . ARS states' elements contain a number of cells, which are first-order terms annotated with a type τ . Cells of type **Bool** and **Int** are interpreted in SMT as Booleans and integers, while cells of the remaining types are encoded as uninterpreted constants in the constants encoding; the arrays encoding approach is discussed in Section 3. Cells are referred to via the notation c_{name} or c_{index} , and they can be seen as both KerA^+ constants and first-order terms in SMT. An arena is a directed acyclic graph $\mathcal{A} = (\mathcal{V}, \mathcal{E})$, with $\mathcal{V}$ being a finite set of cells and $\mathcal{E} \subseteq \mathcal{V} \times (1..|\mathcal{V}|) \times \mathcal{V}$ being a set of relations between the cells in $\mathcal{V}$. Every relation between cells is represented by an arena edge of form (c_a, i, c_b) , also written $c_a \xrightarrow{i} c_b$, with no duplicates, i.e., for every pair $(c_{a_1}, i_1, c_{b_1}), (c_{a_2}, i_2, c_{b_2}) \in \mathcal{E}$ we have that $c_{a_1} = c_{a_2} \wedge c_{b_1} \neq c_{b_2}$ implies $i_1 \neq i_2$.

and no gaps in the relation indexes, i.e., for every edge (c_a, i, c_b) and index $j \in 1..(i-1)$ we have that $\exists c_c \in \mathcal{V} . (c_a, j, c_c)$. A binding is a partial function from KerA^+ variables to $\mathcal{V}$ of $\mathcal{A}$, i.e., a mapping from variables to cells. Finally, Φ is a formula in the SMT fragment supported by the ARS and the target SMT solver, e.g., the quantifier-free uninterpreted functions and non-linear arithmetics (QF_UFNIA) fragment supported by the constants encoding.

A series of n reduction steps has the form $s_0 \rightsquigarrow \dots \rightsquigarrow s_n$, with each step generating state s_{i+1} for state s_i , $0 \leq i < n$, by applying a reduction rule. The initial state $s_0 = (e_0, \mathcal{A}_0, \nu_0, \Phi_0)$ has e_0 as the initial KerA^+ specification, $\mathcal{A}_0 = (\emptyset, \emptyset)$, ν_0 containing no mappings, and $\Phi_0 = \text{true}$. The reduction steps end upon reaching a state $s_n = (e_n, \mathcal{A}_n, \nu_n, \Phi_n)$, with e_n being a single cell $c \in \mathcal{V}_n$ and $\mathcal{A}_n = (\mathcal{V}_n, \mathcal{E}_n)$. Below we give two examples of rules.

Integer literal reduction. One of the simplest rules has an integer literal num being rewritten into a cell c_{num} . This cell is added to the arena and a constraint equating c_{num} to the literal is conjoined with Φ ; we use vertical lines to separate state elements and commas to indicate additions to $\mathcal{A}$ and conjunctions to Φ .

$$\frac{\langle num : \text{Int} \mid \mathcal{A} \mid \nu \mid \Phi \rangle \quad num \text{ is one of } 0, 1, -1, \dots}{\langle c_{num} \mid \mathcal{A}, c_{num} : \text{Int} \mid \nu \mid \Phi, c_{num} = num \rangle} \text{ (INT)}$$

The descriptions of rules can be given as inferences, with the premisses above the bar and the resulting state below it. Inferences, although reasonable to express rules such as INT, are not suitable to give the intuition about how more complex rules work. In light of this, we will use a simplified notation moving forward. We inline inferences as $\rightsquigarrow$ and omit nonessential information, e.g., propagated values. Below we can see rule INT in this simplified format. Note that only $\mathcal{A}$ and Φ updates are shown, without propagating them, and that ν is omitted.

$$\begin{array}{c} num : \text{Int} \\ num \text{ is one of } 0, 1, -1, \dots \end{array} \rightsquigarrow c_{num} \mid c_{num} : \text{Int} \mid c_{num} = num \quad \text{(Int)}$$

Picking. To pick a cell out of n cells we use an oracle θ , as per rule **FromBasic**. In addition to the FROM ... BY θ expression, this rule requires that all pickable cells are of the same basic type τ , e.g., **Int**. The resulting state has a new cell c_{pick} , which is equated to one of the n cells if $1 \leq \theta \leq n$ and is unconstrained otherwise. Picking among cells representing data structures, e.g., sets, can be done via a more general version of rule **FromBasic**, which we omit for brevity.

$$\begin{array}{c} \text{FROM } c_1, \dots, c_n \text{ BY } \theta : \tau \\ \tau \text{ is basic and } c_1 : \tau, \dots, c_n : \tau \end{array} \rightsquigarrow c_{pick} \mid c_{pick} : \tau \mid \bigwedge_{1 \leq i \leq n} (\theta = i \rightarrow c_{pick} = c_i) \quad \text{(FromBasic)}$$

2.4 SMT Theory of Arrays

The theory of arrays provides a natural way to encode data structures and is thus a prime candidate as an encoding target for TLA^+ constructs. Here we present the theory's operators relevant for our work, further details can be found in [8].

Given the set of sorts S , containing one sort s_τ for each type τ in KerA^+ , an array sort s_{τ_1, τ_2} has the form $s_{\tau_1} \Rightarrow s_{\tau_2}$, with $s_{\tau_1} \in S$ being its *index sort* and $s_{\tau_2} \in S$ being its *value sort*. Each array sort is supported by two basic operators, *select* : $(s_{\tau_1} \Rightarrow s_{\tau_2}, s_{\tau_1}) \rightarrow s_{\tau_2}$, which handles array access at a given index, and *store* : $(s_{\tau_1} \Rightarrow s_{\tau_2}, s_{\tau_1}, s_{\tau_2}) \rightarrow s_{\tau_1} \Rightarrow s_{\tau_2}$, which updates an array for a given index and value. For brevity, we will write *select*(a, i) as $a[i]$ in the remainder of the manuscript. Regarding equality between arrays, different interpretations are possible. We use arrays with extensionality [25], which are considered equal if they contain the same values in the same entries. Extensionality is formally defined as $\forall a, b : s_{\tau_1} \Rightarrow s_{\tau_2} . a = b \vee \exists i : s_{\tau_1} . a[i] \neq b[i]$. For access and update, consistency is ensured by the following property:

$$\forall a : s_{\tau_1} \Rightarrow s_{\tau_2}, i : s_{\tau_1}, j : s_{\tau_1}, v : s_{\tau_2} .$$

$$\underbrace{\text{store}(a, i, v)[i] = v}_{\text{access consistency}} \wedge \underbrace{(i = j \vee \text{store}(a, i, v)[j] = a[j])}_{\text{update consistency}}$$

In addition to *select* and *store*, the theory of arrays can be extended with other operators, two of which are map_f and K_{s_τ} , whose signatures are shown below. The map_f operator applies a n -ary function $f : (s_{\tau_1}, \dots, s_{\tau_n}) \rightarrow s_\tau$ to the values stored in each index of its array arguments, producing a new array whose values are the result of the function application, i.e., map_f is the pointwise array extension of f . The K_{s_τ} operator produces a constant array, with all its values being the constant provided as argument. The properties defining the behaviour of these two operators are shown after their signatures.

$$\text{map}_f : (s_\tau \Rightarrow s_{\tau_1}, \dots, s_\tau \Rightarrow s_{\tau_n}) \rightarrow s_\tau \Rightarrow s_{\tau_f} \quad K_{s_\tau} : s_{\tau_{\text{const}}} \rightarrow s_\tau \Rightarrow s_{\tau_{\text{const}}}$$

$$\forall a_1 : s_\tau \Rightarrow s_{\tau_1}, \dots, a_n : s_\tau \Rightarrow s_{\tau_n}, i : s_\tau . \text{map}_f(a_1, \dots, a_n)[i] = f(a_1[i], \dots, a_n[i])$$

$$\forall i : s_{\tau_1}, v : s_{\tau_2} . K_{s_{\tau_1}}(v)[i] = v$$

The *select* and *store* operators are part of theory of arrays with extensionality defined in version 2.6 of the SMT-LIB standard [3]. Other operators are provided on a solver-by-solver basis, e.g., Z3 [7] supports both map_f and K_{s_τ} , while CVC5 [2] supports K_{s_τ} ; SMT-LIB updates may add them to the standard.

3 Encoding TLA+ using Arrays

Our goal is to encode TLA^+ data structures in a structure-preserving way. To do this, we use arrays to represent the main components of TLA^+ , sets and functions, as SMT constraints. We follow the ARS structure described in Section 2.3, but update the reduction rules handling sets and functions. The remaining TLA^+ constructs, e.g., tuples, are represented as per the constants encoding.

The two efficiency benefits of the arrays encoding are the ease of access of data structures and the possibility of using SMT equality. The first benefit can be easily understood by the use of SMT *select*, which allows us to check a stored value by using a single constraint, in contrast to the amount of constraints

used in the constants encoding, which is linear in the size of data structures' overapproximation. The second benefit affects the comparison of data structures, which can be done via a single SMT equality for sets and functions in the arrays encoding, since these structures are represented by a single SMT term, while the constants encoding requires a number of constraints that is quadratic in the size of data structures' overapproximation. A summary can be seen in Table 1. We first describe how to encode sets and functions, and then present the correctness argument for the reduction to arrays.

3.1 Encoding TLA+ Sets using Arrays

We use arrays to encode TLA⁺ sets as characteristic functions, i.e., a set of type τ is represented by an array of sort $s_\tau \Rightarrow \text{Bool}$. Set membership is encoded by storing **true** or **false** on a given array index. The reduction rules used to handle the main set operators are presented below.

Set Enumeration. The simplest way to create a set is to enumerate its elements. Rule **Enum** reduces an explicit set of cells to a fresh cell c_{set} , whose edges link it to its elements; $c_{set} \rightarrow c_1, \dots, c_n$ is a shorthand for $c_{set} \xrightarrow{1} c_1, \dots, c_{set} \xrightarrow{n} c_n$. There is no guarantee that the enumerated elements are unique, thus the arena may contain edges to repeated elements.

$$\{c_1, \dots, c_n\} : \text{Set}[\tau] \mapsto c_{set} \mid c_{set} : \text{Set}[\tau], c_{set} \rightarrow c_1, \dots, c_n \mid \text{EnumCtr} \quad (\text{Enum})$$

The constraints *EnumCtr* added by the arrays encoding create an empty set, by using a constant array with the value **false**, $\perp$, and updates the array by storing **true**, $\top$, on the appropriate indexes. The array resulting from the last update, $a_{c_{set}}^n$, is then equated to c_{set} . Since cells representing repeated elements lead to updates to the same index, we encode standard sets, in contrast the constants encoding, which encodes multisets due to the arena imprecision; multisets lead to multiple constraints being generated to encode membership of a single element.

$$\underbrace{a_{c_{set}}^0 = K_\tau(\perp)}_{\text{empty set}} \wedge \underbrace{\bigwedge_{1 \leq i \leq n} a_{c_{set}}^i = \text{store}(a_{c_{set}}^{i-1}, c_i, \top)}_{\text{set updates}} \wedge \underbrace{c_{set} = a_{c_{set}}^n}_{\text{cell equality}} \quad (\text{EnumCtr})$$

Although the amount of constraints generated by the arrays encoding to model set enumeration is equal to that of the constants encoding, it has the benefit of generating a defined interpretation for c_{set} , the array $a_{c_{set}}^n$, which is not present in the constants encoding. This has a significant impact on set membership and cell equality, as described below.

Set Membership. The checking of a membership relation $c_x \in c_{set}$, given the presence of the arena edges $c_{set} \rightarrow c_1, \dots, c_n$ and $1 \leq x \leq n$, is straightforward. A single fresh cell of Boolean type is introduced and is equated to $c_{set}[c_x]$.

Cell Equality. The constraints generated by encoding set membership and many other constructs assume that cells can be compared. When this is not directly the case the equalities are cached in preparation. For example, if a set of n tuples c_t of size two is being equated, the constraints $c_{t_i} = c_{t_j} \leftrightarrow c_{t_i}^1 = c_{t_j}^1 \wedge c_{t_i}^2 = c_{t_j}^2$, with $1 \leq i \leq n$ and $1 \leq j \leq n$, are added to Φ ; here we use c_t^1 and c_t^2 to represent the values of the 2-tuple. The need for this caching of equalities only arises when data structures encoded as uninterpreted constants are compared. For the remaining rules we assume that caching was done, if needed, and cells can be compared via direct equality.

Table 1: Amount of constraints generated by each SMT encoding to model the main TLA⁺ constructs.

Construct	Arrays	Constants
Set enumeration	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set membership	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Set equality	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
Set filter	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Set map	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Fun. definition	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Fun. domain	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Fun. equality	$\mathcal{O}(1)$	$\mathcal{O}(n^2)$
Fun. update	$\mathcal{O}(1)$	$\mathcal{O}(n)$
Fun. application	$\mathcal{O}(n)$	$\mathcal{O}(n)$

Set Filter. In TLA⁺, the elements of a set S can be filtered by a predicate p via the expression $\{x \in S : p\}$. This expression will create a set F which contains only the elements of S that satisfy p , e.g., $\{x \in \{-1, 0, 1\} : x \geq 0\} = \{0, 1\}$. Rule **Filter** reduces a filter to a new set cell, c_F , whose arena overapproximation contains the elements of S , but whose constraints ensure that only filtered elements are members of F ; $p[y/x]$ means that x is replaced by y in p and parentheses indicate the application of another rule, the predicate resolution rule in this case.

$$\begin{aligned}
\{x \in c_S : p\} : \text{Set}[\tau] \text{ and } c_S \rightarrow c_1, \dots, c_n \\
\mapsto (p[c_1/x] : \text{Bool}, \dots, p[c_n/x] : \text{Bool} \mapsto c_1^p, \dots, c_n^p) \\
\mapsto c_F \mid c_F : \text{Set}[\tau], c_F \rightarrow c_1, \dots, c_n \mid \text{FilterCtr}
\end{aligned}
\tag{Filter}$$

The constraints added use an array $a_{c_F}^0$ initially unconstrained, i.e., the values mapped by all the indexes of $a_{c_F}^0$ are unconstrained, as opposed to $a_{c_{set}}^0$ in *EnumCtr*. The values of $a_{c_F}^0$ mapped by indexes $c_1, \dots, c_n$ are constrained by $c_1^p, \dots, c_n^p$ via array access, i.e., $a_{c_F}^0[c_i]$ is asserted to be true or false based on c_i^p , with $1 \leq i \leq n$. We then apply pointwise conjunction to c_S and $a_{c_F}^0$ via the map_f SMT operator; we go from a_F^0 to a_F^n to keep the array index in step with the arena overapproximation. Indexes whose values were false in S remain so in F , and indexes whose values were true in S store the filter's predicate evaluation.

$$\underbrace{\bigwedge_{1 \leq i \leq n} \text{ite}(c_i^p, a_{c_F}^0[c_i], \neg a_{c_F}^0[c_i])}_{\text{predicate-based constraining}} \wedge \underbrace{a_{c_F}^n}_{\text{pointwise conjunction}} = \underbrace{map_\wedge(c_S, a_{c_F}^0)}_{\text{cell equality}} \wedge c_F = a_{c_F}^n \quad (\text{FilterCtr})$$

Both encodings generate a linear amount of constraints, since n $p[c_i/x]$ predicates have to be considered. Unlike with *EnumCtr*, *FilterCtr* does not contain many *store* operations, due to the usage of map_f . This avoids the need to create intermediary arrays, and is not possible in *EnumCtr* due to its constant array.

Set Map. The expression $\{e : x \in S\}$ can be used to construct a set M from a set S , having all the elements of M as $e[y/x]$, with $y \in S$. For example, the expression $\{x \div 5 : x \in \{4, 5, 6\}\}$ yields the set $\{0, 1\}$, with $\div$ denoting standard integer division. To reduce set map we use rule **Map**.

$$\begin{aligned} \{e : x \in c_S\} : \text{Set}[\tau] \text{ and } c_S \rightarrow c_1, \dots, c_n \\ \mapsto (e[c_1/x] : \tau, \dots, e[c_n/x] : \tau \mapsto c_1^e, \dots, c_n^e) \\ \mapsto c_M \mid c_M : \text{Set}[\tau], c_M \rightarrow c_1^e, \dots, c_n^e \mid \text{MapCtr} \end{aligned} \quad (\text{Map})$$

The constraints added in rule **Map** are similar to those added in rule **Enum**. The difference between them is that set enumeration precisely defines the elements to be added to the new set cell, while set map is based on an existing set cell, which is a set overapproximation. Due to this, membership in M has to be guarded by membership in S , leading to a linear amount of constraints being generated.

$$\underbrace{a_{c_M}^0 = K_\tau(\perp)}_{\text{empty set}} \wedge \underbrace{\bigwedge_{1 \leq i \leq n} \text{ite} \left(\begin{array}{c} c_S[c_i], \\ a_{c_M}^i = \text{store}(a_{c_M}^{i-1}, c_i^e, \top), \\ a_{c_M}^i = a_{c_M}^{i-1} \end{array} \right)}_{\text{set updates}} \wedge \underbrace{c_M = a_{c_M}^n}_{\text{cell equality}} \quad (\text{MapCtr})$$

3.2 Encoding TLA⁺ Functions using Arrays

We use arrays to encode TLA⁺ functions directly as functions themselves. To do this, arrays are used in their general format, with a function $f : s_{\tau_1} \rightarrow s_{\tau_2}$ being encoded as an array of sort $s_{\tau_1} \Rightarrow s_{\tau_2}$. Since functions with a finite domain can rely on infinite sorts, e.g., the integer numbers, the encoding of each function also includes constraints defining its domain set, by means of the rules described in the previous section; the result of a function application to a value outside its domain is undefined in TLA⁺. This approach allows us to generate SMT constraints that follow directly from TLA⁺, making the encoding not only more efficient, but also more natural to describe. In contrast, the constants encoding represents functions explicitly as sets of pairs of form $\{\langle x, f(x) \rangle : x \in \text{DOMAIN } f\}$. Due to this, its function manipulation relies on set manipulation, e.g., function comparison is encoded as set comparison, leading to a quadratic amount of constraints. The reduction rules used to handle functions are presented below.

Function Definition. The definition of a function in TLA⁺ is an expression of the form $[x \in S \mapsto e]$, which maps every domain value v to the expression $e[v/x]$. This definition is similar to that of set map $\{e : x \in S\}$, and thus generates constraints in a similar fashion to rule **Map**. The main difference is that the evaluations of the expression $e[v/x]$ are stored as array values, rather than array indexes, i.e., function definition uses $\text{store}(a, v, e[v/x])$ and set map uses $\text{store}(a, e[v/x], \top)$, with v being a value in the function's domain or the set being mapped. Every encoded function has a single argument, with multiple arguments being rewritten as tuples in preprocessing.

Unlike with set cells, a function cell c_F in the arena does not directly point to its values, with the arrays encoding adding two edges to c_F , $c_F \xrightarrow{1} c_{F_{dom}}$ and $c_F \xrightarrow{2} c_{F_{pairs}}$. Cell $c_{F_{dom}}$ represents the function's domain and cell $c_{F_{pairs}}$ represents the set of pairs $\{\langle x, f(x) \rangle : x \in \text{DOMAIN } f\}$. Cell $c_{F_{pairs}}$, despite being in the arena, has no SMT constraints modelling it in the arrays encoding, with its sole purpose being to help propagate the arena edges of the function's codomain elements.

Function Domain. Accessing a function's domain is trivial in the arrays encoding, since the domain set is generated during function definition. This results in a simple access to the array representing the domain.

Function Update. The update of a TLA⁺ function f is done by changing the result of applying f to an argument arg , $f[arg]$, to be a given value v , via the expression $[f \text{ EXCEPT! } [arg] = v]$. The update will produce a new function g which is identical f , except that $g[arg] = v$ if $arg \in \text{DOMAIN } f$. The arrays encoding generates a single array update constraint in this case.

Function Application. The application of a function to an argument arg is conceptually simple, but is quite intricate to realize, as can be seen in rule **FunApp**. The arrays encoding uses an oracle to check that c_{arg} is in the domain and to gather the arena edges of c_{res} . The *FunAppCtr* constraints ensure that the oracle chooses the correct index and equates the result cell to an array access on c_F . Note that the value of c_{res} comes directly from the function application expression itself, with the oracle only been needed to gather the arena edges of c_{res} , if $m > 0$, via c^p . The need for an oracle is restricted to functions whose codomain contain structured data, e.g., $f : \text{Int} \rightarrow \text{Set}[\text{Int}]$. If this is not the case, e.g., $g : \text{Int} \rightarrow \text{Int}$, rule **FunApp** is simplified and *FunAppCtr* becomes $c_{res} = c_F[c_{arg}]$.

$$\begin{aligned}
& c_F[c_{arg}] : \tau \text{ and } c_F \xrightarrow{1} c_{F_{dom}} \rightarrow c_1^d, \dots, c_n^d \text{ and } c_F \xrightarrow{2} c_{F_{pairs}} \rightarrow c_1^p, \dots, c_n^p \\
& \quad \mapsto \text{(FROM } c_1^p, \dots, c_n^p \text{ BY } \theta : \langle \tau_{arg}, \tau \rangle \mid \theta : \text{Int} \mid 0 \leq \theta \leq n \mapsto c^p) \\
& \quad \text{and } c^p[2] \rightarrow c_1, \dots, c_m \\
& \quad \mapsto c_{res} \mid c_{res} : \tau, c_{res} \rightarrow c_1, \dots, c_m \mid \text{FunAppCtr} \\
& \hspace{25em} \text{(FunApp)} \\
& \underbrace{\bigwedge_{1 \leq i \leq n} \wedge \left(\begin{array}{l} (\theta = i \rightarrow c_{arg} = c_i^d \wedge c_{F_{dom}}[c_i^d]) \\ (\theta = 0 \rightarrow c_{arg} \neq c_i^d \vee \neg c_{F_{dom}}[c_i^d]) \end{array} \right)}_{\text{oracle constraining}} \wedge \underbrace{c_{res} = c_F[c_{arg}]}_{\text{cell equality}} \quad \text{(FunAppCtr)}
\end{aligned}$$

3.3 Correctness of the Reduction to Arrays

Correctness of the ARS is given by four properties: finiteness of the models, compliance to the target SMT theories, termination of any reduction sequence, and soundness of the reductions. These properties have their correctness sketched for the constants encoding in [13], with detailed proofs present in [26]. Since we rely on the existing ARS and restrict our changes to mainly affect constraint generation, we have the same degree of overapproximation and the correctness

arguments made for the constants encoding are in large part valid for the arrays encoding. We present below the definition of a KerA⁺ model and detail, for each property, how the use of arrays affects the correctness arguments and how they can be adjusted to remain valid.

Models. Every satisfiable KerA⁺ formula has a model $\mathcal{M} = \langle \mathcal{D}, \mathcal{I} \rangle$, where $\mathcal{D}$ is the model domain, consisting of a disjoint union of sets $\mathcal{D}_1, \dots, \mathcal{D}_n$, with $\mathcal{D}_i$, $1 \leq i \leq n$, containing the values for type τ_i , and $\mathcal{I}$ is the model interpretation, consisting of assignments of domain values to KerA⁺ constants. Models are used to access cell values, with the value of a KerA⁺ expression e in model $\mathcal{M}$ being $\llbracket e \rrbracket^{\mathcal{M}}$. In $s_{\text{before}} \rightsquigarrow s_{\text{after}}$, we go from $\mathcal{M}_{\text{before}}$ to $\mathcal{M}_{\text{after}}$, with $\mathcal{M}_{\text{after}}$ containing the interpretation of additional constants and being thus an extension of $\mathcal{M}_{\text{before}}$.

Finiteness. This property states that every interpretation of a KerA⁺ expression is defined only over finite values. Its proof is derived from the finiteness of the elements being modelled. In the arrays encoding, we potentially use arrays with infinite sorts, e.g., the integers, but all SMT interpretations that can be derived from such arrays are finite, since we encode only finite TLA⁺ data structures. This guarantees finiteness of all KerA⁺ models in the arrays encoding.

Theory Compliance. This property states that any sequence of states $s_0 \rightsquigarrow \dots \rightsquigarrow s_n$ has the formulas Φ_i , $1 \leq i \leq n$, in the first-order logic fragment containing only quantifier-free expressions over uninterpreted functions and integer arithmetic. Its proof is done by induction on the constraints generated. The constraint Φ_0 is always true and is thus trivially compliant. The inductive case is proved by showing that the constraint added by each rule are compliant. The rules in the arrays encoding only add array constraints, in addition to constraints supported by the constants encoding, so theory compliance is straightforward to guarantee.

Termination. This property states that every sequence of ARS reductions is finite, i.e., the reduction process always terminates. Its proof is based on ensuring that every rule r applied to a given state s_{before} yields a state s_{after} with e_{after} being smaller than e_{before} . An expression's length is given based on the length of its sub-expressions. The arrays encoding mainly changes constraint generation, and in the cases where rules are slightly modified they generate resulting expressions of the same size, thus guaranteeing termination.

Soundness. This property is described in Theorem 1. Both e and Φ are KerA⁺ expressions, but Φ is in the first-order logic fragment supported by SMT solvers. Fundamentally, the ARS is rewriting a formula to forward it to the solver. The soundness proof consists of case analysis of each reduction rule to establish that $e_{\text{before}} \wedge \Phi_{\text{before}}$ is equisatisfiable to $e_{\text{after}} \wedge \Phi_{\text{after}}$, no matter the rule applied in $s_{\text{before}} \rightsquigarrow s_{\text{after}}$. The case analysis, which describes how e_{after} and Φ_{after} can be derived from e_{before} and Φ_{before} for each rule, relies on six invariants of the reduction system. Three invariants, 1, 3, and 4, are encoding independent, and thus are the same as in [13], the remaining three, 2, 5, and 6, are changed due to

the new representation of sets and functions. Below we show all six invariants, with the modifications needed to guarantee soundness for the arrays encoding.

Theorem 1. *Let $s_0 \rightsquigarrow \dots \rightsquigarrow s_n$ be a sequence of states produced by the ARS, with $s_i = \langle e_i \mid \mathcal{A}_i \mid \nu_i \mid \Phi_i \rangle$ and $1 \leq i \leq n$. Assume that e_0 is a formula, i.e., it has type `Bool`. Then e_0 is satisfiable iff the conjunction $e_n \wedge \Phi_n$ is satisfiable.*

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

Invariant 2 (arena membership) *In every reachable state $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ of the ARS, every cell c in either the expression e or the formula Φ is also in $\mathcal{A}$.*

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

Invariant 4 (overapproximation) *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its model. Assume that c_{set} is a set cell in the arena $\mathcal{A}$ and that $c_{\text{set}} \rightarrow c_1, \dots, c_n$ are edges in $\mathcal{A}$, for some $n \geq 0$. Then, it holds that $\llbracket c_{\text{set}} \rrbracket^{\mathcal{M}} \subseteq \{\llbracket c_1 \rrbracket^{\mathcal{M}}, \dots, \llbracket c_n \rrbracket^{\mathcal{M}}\}$.*

Invariant 5 (function domain) *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS. Assume that c_f is a function cell of type $s_{\tau_1} \rightarrow s_{\tau_2}$ in the arena $\mathcal{A}$. Then, there is a cell c_{dom} of type $s_{\text{Set}[\tau_1]}$ such that $c_f \xrightarrow{1} c_{\text{dom}}$.*

Invariant 6 (domain reduction) *Let $\langle e \mid \mathcal{A} \mid \nu \mid \Phi \rangle$ be a reachable state of the ARS, and $\mathcal{M}$ be its model. Assume that c_f is a function cell and that $c_f \xrightarrow{1} c_{F_{\text{dom}}}$ is in the arena $\mathcal{A}$. Then, it follows that $\llbracket c_{F_{\text{dom}}} \rrbracket^{\mathcal{M}} = \llbracket \text{DOMAIN } f \rrbracket^{\mathcal{M}}$.*

As described in sections 3.1 and 3.2, arrays precisely model TLA⁺ sets and functions. The handling of sets revolves around membership constraints of form $c_{\text{set}}[c_i]$, which can be set to `true` or `false` via *store*. Regarding functions, function application and update are trivially equivalent to array access and update. The more elaborate array operators also have a counterpart in TLA⁺. Constant arrays are equivalent to a function definition for which all range values are the same constant, and array map is equivalent to set map. These equivalences explain how the changes in the arrays encoding do not invalidate the case analysis of the reduction rules used to prove Theorem 1, thus guaranteeing soundness.

4 Evaluation

In order to evaluate the performance impact of the arrays-based encoding, we implemented it in the APALACHE model checker, which currently supports the constants encoding. Given a TLA⁺ specification containing a property P , APALACHE is capable of performing bounded model checking up to a length k and, if P is an inductive invariant, it can check if the property holds with an unbounded

length. In both modes, APALACHE checks if the SMT formula encoding the specification is satisfiable when conjoined with $\neg P$, and if that is the case a counterexample (CEX) in the form of a trace is produced using the arena information and the satisfiable assignment provided by the SMT solver. Our implementation adds new reduction rules to APALACHE, which can be enabled via a CLI flag. When enabled, these rules replace the existing ones encoding sets and functions, as described in Section 3. In addition, we also extended APALACHE’s CEX generation to handle assignments to SMT formulas containing arrays. We use Z3 [7] as our back-end solver. APALACHE is open-source and freely available³.

We performed a number of experiments using APALACHE and the explicit-state model checker TLC. For APALACHE, we evaluated both its existing constants encoding and two versions of the arrays encoding we propose, called *arrays* and *funArrays*. The *arrays* version encodes both TLA^+ sets and functions as arrays, while the *funArrays* version encodes only TLA^+ functions as arrays. The purpose of having two versions of our encoding is to evaluate the impact of encoding sets and functions as arrays separately. Our evaluation setup consisted of a machine with 64 AMD EPYC 7452 processors and 256 GB of memory. We first present the benchmarks used and then discuss the results obtained.

4.1 Benchmarks

We consider the TLA^+ specifications of three asynchronous protocols as benchmarks. The first benchmark is a specification of the asynchronous Byzantine agreement protocol by Bracha and Toueg [5], showed in a simplified version in Figure 2, to which we refer as *aba*. The second benchmark is a specification of the consensus algorithm with Byzantine faults in one communication step by Dobre and Suri [9], to which we refer as *cab*. The third benchmark is a specification of the asynchronous non-blocking atomic commitment protocol by Guerraoui [12], to which we refer as *nac*. The common use of *aba* and *cab* is in replication scenarios with $N = 3F + 1$ replica nodes to tolerate F failures, while the *nac* protocol is typically used for partitioned databases. The specifications are available online⁴.

4.2 Results

For each specification we check a variation of the agreement property. The results are shown in Figure 4. We can see that both *arrays* and *funArrays* scale in performance better than the constants encoding, with an order of magnitude improvement for some instances. It is also worth pointing out that *arrays* and *funArrays* were able to reach a result before the time limit in 29 and 28 instances, respectively, while the constants encoding was able to do so in only 20 instances. In regards to TLC, it performed worse than the three APALACHE encodings in the nontrivial cases, only reaching a result before the time limit in 8 instances.

³ Available at <https://github.com/informalsystems/apalache>

⁴ Available at <https://github.com/informalsystems/apalache-bench>

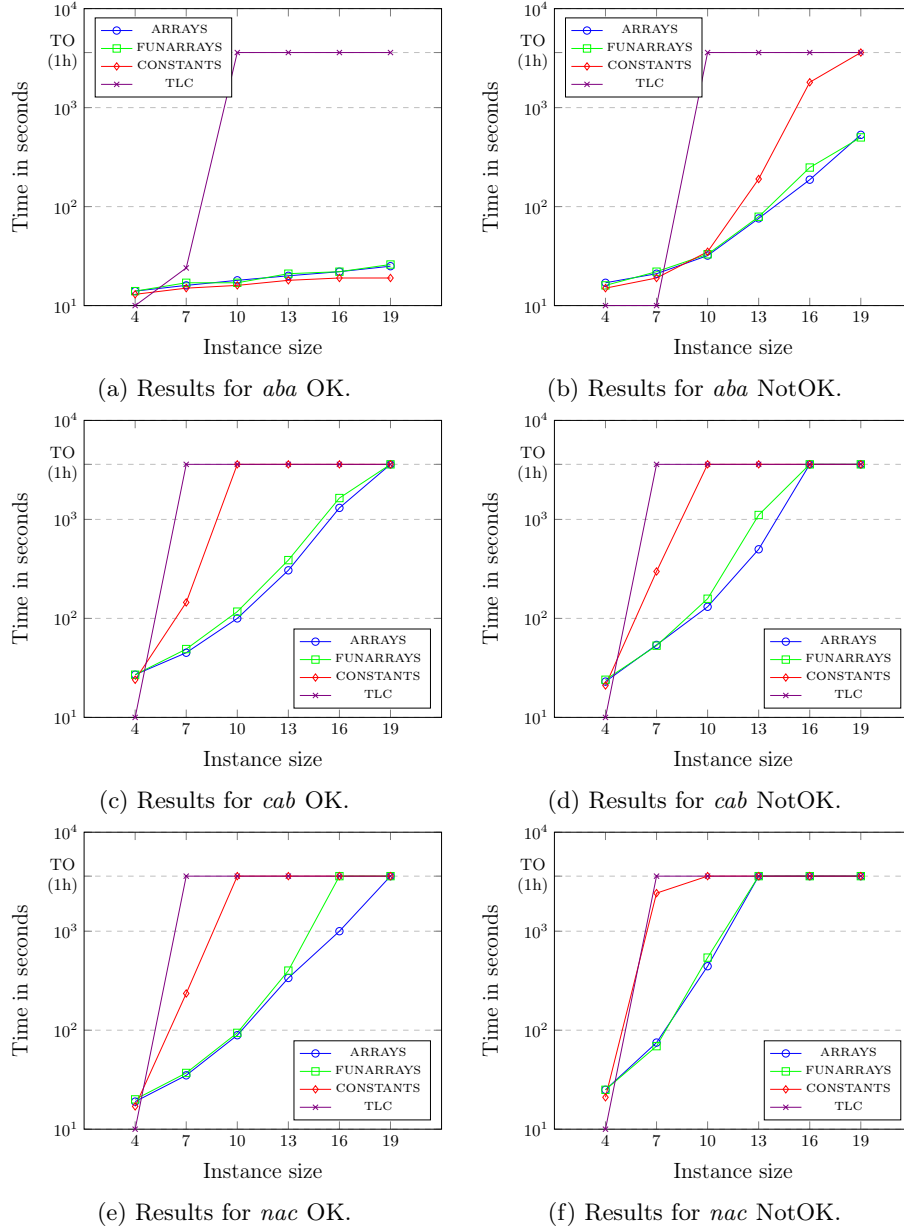


Fig. 4: Time in checking agreement for *aba*, *cab*, and *nac*. Specifications were ran in two configurations, one in which agreement is expected to hold (OK) and one in which it is not (NotOK). Instance size stands for the number of nodes used, and the time is given in seconds in logarithmic scale; Timeout (TO) is 1 hour.

5 Related Work

An extensive discussion of works related to symbolic model checking for TLA^+ can be found in [13]. Here we focus exclusively on closely related publications. The IVy Prover [20] was designed to tackle verification of distributed algorithms with a decidable fragment of relational first-order logic. Some distributed algorithms, such as the one in Figure 2, cannot be directly expressed in this fragment however, due to the use of power sets and set cardinalities. Recent efforts have focused on offering support to reason about set cardinalities [4], but limitations remain. Cut-off based techniques to automatically infer invariants of distributed algorithms in the IVy language, such as relational abstractions of Paxos and two-phase commit, have been recently proposed [10,11]. Similar benchmarks are used in [22] to infer generalized invariants from finite instances of TLA^+ and semi-automatically prove invariants with TLAPS. Specifications of fault-tolerant distributed algorithms encoded as threshold automata can be efficiently verified with ByMC [15,24]. The manual rewriting of an algorithm into threshold automata is, however, usually beyond the skills of a typical TLA^+ user. The work closest to ours involves the use of SMT arrays to encode EventB and TLA^+ specifications in ProB [21]. The focus on ProB aims at handling infinite data structures, in contrast to our choice to work with bounded overapproximations. Reasoning about infinite domains implies the use of quantifiers, which prevents the use of efficient decision procedures available for the decidable fragment of SMT, with this approach been shown to underperform when compared against APALACHE in checking the benchmarks from [13]. An important last point to mention is that CVC5 has its own non-standard SMT theory of sets [1]. This theory, however, cannot currently handle nested sets, which is a very commonly used TLA^+ construct. It remains as a viable alternative to the SMT theory of arrays for the encoding of flat sets, but whose use implies important restrictions to the input language and, consequentially, to practical application.

6 Conclusions

We propose an encoding of the main TLA^+ constructs into the SMT theory of arrays, with the goal of providing the SMT solver with the structural information it needs to efficiently reach a solution. We implemented our encoding into the APALACHE model checker and our evaluation indicates that our arrays-based encoding provides a significant performance improvement when compared against APALACHE’s existing SMT encoding and the explicit-state model checker TLC. Encoding the remaining TLA^+ constructs in a structure-preserving way, be it via SMT arrays or algebraic datatypes, remains an interesting research avenue.

Acknowledgements Rodrigo Otoni and Natasha Sharygina’s work was supported by the Swiss National Science Foundation, via grants 200021_197353 and 200021_185031, respectively. Igor Konnov and Jure Kukovec’s work was supported by the Interchain Foundation. The authors thank Shon Feder for his kind assistance in preparing the evaluation infrastructure.

References

1. Bansal, K., Reynolds, A., Barrett, C., Tinelli, C.: A New Decision Procedure for Finite Sets and Cardinality Constraints in SMT. In: Proceedings of the 8th International Joint Conference on Automated Reasoning. pp. 82–98 (2016)
2. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: CVC5: A Versatile and Industrial-Strength SMT Solver. In: Proceedings of the 28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442 (2022)
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep. (2021), Available at <https://smtlib.cs.uiowa.edu>
4. Berkovits, I., Lazic, M., Losa, G., Padon, O., Shoham, S.: Verification of Threshold-Based Distributed Algorithms by Decomposition to Decidable Logics. In: Proceedings of the 31st International Conference on Computer Aided Verification. pp. 245–266 (2019)
5. Bracha, G., Toueg, S.: Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM* **32**(4), 824–840 (1985)
6. Chaudhuri, K., Doligez, D., Lamport, L., Merz, S.: The TLA+ Proof System: Building a Heterogeneous Verification Platform. In: Proceedings of the 7th International Colloquium on the Theoretical Aspects of Computing. p. 44 (2010)
7. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340 (2008)
8. De Moura, L., Bjørner, N.: Generalized, Efficient Array Decision Procedures. In: Proceedings of the 9th International Conference on Formal Methods in Computer-Aided Design. pp. 45–52 (2009)
9. Dobre, D., Suri, N.: One-step Consensus with Zero-Degradation. In: Proceedings of the 36th International Conference on Dependable Systems and Networks. pp. 137–146 (2006)
10. Goel, A., Sakallah, K.: On Symmetry and Quantification: A New Approach to Verify Distributed Protocols. In: Proceedings of the 13th NASA Formal Methods International Symposium. p. 131–150 (2021)
11. Goel, A., Sakallah, K.A.: Towards an Automatic Proof of Lamport’s Paxos. In: Proceedings of the 21st Conference on Formal Methods in Computer-Aided Design. pp. 112–122 (2021)
12. Guerraoui, R.: On the Hardness of Failure-Sensitive Agreement Problems. *Information Processing Letters* **79**(2), 99–104 (2001)
13. Konnov, I., Kukovec, J., Tran, T.H.: TLA+ Model Checking Made Symbolic. *Proc. ACM Program. Lang.* **3**(OOPSLA), 123:1–123:30 (2019)
14. Konnov, I., Kuppe, M., Merz, S.: Specification and Verification with the TLA+ Trifecta: TLC, Apalache, and TLAPS. In: Proceedings of the 11th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation. pp. 88–105 (2022)
15. Konnov, I., Lazić, M., Stoilkovska, I., Widder, J.: Tutorial: Parameterized Verification with Byzantine Model Checker. In: Proceedings of the 40th International Conference on Formal Techniques for Distributed Objects, Components, and Systems. pp. 189–207 (2020)

16. Kukovec, J., Tran, T.H., Konnov, I.: Extracting Symbolic Transitions from TLA+ Specifications. *Science of Computer Programming* **187**, 102361 (2020)
17. Lamport, L.: *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc. (2002)
18. Merz, S., Vanzetto, H.: Encoding TLA+ into unsorted and many-sorted first-order logic. *Science of Computer Programming* **158**, 3–20 (2018)
19. Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., Deardeuff, M.: How Amazon Web Services uses Formal Methods. *Communications of the ACM* **58**(4), 66–73 (2015)
20. Padon, O., McMillan, K.L., Panda, A., Sagiv, M., Shoham, S.: Ivy: Safety Verification by Interactive Generalization. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. pp. 614–630 (2016)
21. Schmidt, J., Leuschel, M.: Improving SMT Solver Integrations for the Validation of B and Event-B Models. In: *Proceedings of the 26th International Conference on Formal Methods for Industrial Critical Systems*. pp. 107–125 (2021)
22. Schultz, W., Dardik, I., Tripakis, S.: Plain and Simple Inductive Invariant Inference for Distributed Protocols in TLA+. In: *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design*. pp. 273–283 (2022)
23. Sivasubramanian, S.: Amazon DynamoDB: A Seamlessly Scalable Non-Relational Database Service. In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. p. 729–730 (2012)
24. Stoilkovska, I., Konnov, I., Widder, J., Zuleger, F.: Verifying Safety of Synchronous Fault-Tolerant Algorithms by Bounded Model Checking. *International Journal on Software Tools for Technology Transfer* **24**(1), 33–48 (2022)
25. Stump, A., Barrett, C., Dill, D., Levitt, J.: A Decision Procedure for an Extensional Theory of Arrays. In: *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. pp. 29–37 (2001)
26. Tran, T.H.: *Symbolic Verification of TLA+ Specifications with Applications to Distributed Algorithms*. Ph.D. thesis, Technische Universität Wien (2023), upcoming thesis
27. Yu, Y., Manolios, P., Lamport, L.: Model Checking TLA+ Specifications. In: *Proceedings of the 10th Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. pp. 54–66 (1999)