Regular Abstractions for Array Systems

CHIH-DUO HONG*, National Chengchi University, Taiwan
ANTHONY W. LIN, University of Kaiserslautern-Landau, Germany and MPI-SWS, Germany

Verifying safety and liveness over array systems is a highly challenging problem. Array systems naturally capture parameterized systems such as distributed protocols with an unbounded number of processes. Such distributed protocols often exploit process IDs during their computation, resulting in array systems whose element values range over an infinite domain. In this paper, we develop a novel framework for proving safety and liveness over array systems. The crux of the framework is to overapproximate an array system as a string rewriting system (i.e. over a finite alphabet) by means of a new predicate abstraction that exploits the so-called indexed predicates. This allows us to tap into powerful verification methods for string rewriting systems that have been heavily developed in the last two decades or so (e.g. regular model checking). We demonstrate how our method yields simple, automatically verifiable proofs of safety and liveness properties for challenging examples, including Dijkstra's self-stabilizing protocol and the Chang-Roberts leader election protocol.

CCS Concepts: • **Theory of computation → Automated reasoning**; **Verification by model checking**; **Program verification**; **Logic and verification**; **Program analysis**.

Additional Key Words and Phrases: Array theory, Regular model checking, Infinite-state model checking, Abstract interpretation, Predicate abstraction, Distributed protocol verification

ACM Reference Format:
Chih-Duo Hong and Anthony W. Lin. 2024. Regular Abstractions for Array Systems. *Proc. ACM Program. Lang.* 8, POPL, Article 22 (January 2024), 29 pages. https://doi.org/10.1145/3632864

1 INTRODUCTION

Over the past few decades, extensive research efforts (e.g. [Cimatti et al. 2021; Felli et al. 2021; Gurfinkel et al. 2016; Ma et al. 2019; Mann et al. 2022]) have been devoted to the verification of *array systems*. Array systems are natural models of sequential programs manipulating linear data structures such as arrays and lists. In addition, they have also been used as convenient abstractions of parameterized concurrent systems with local and shared variables (cf. [Alberti et al. 2017; Bloem et al. 2015]). Specifically, many distributed protocols require each process to maintain a numerical local variable as its process identifier, rendering these protocols suitable to be modeled as array systems.

Despite the amount of work on array systems verification, the problem remains highly challenging. Difficulty arises from the following two aspects, among others: (1) array elements range over an *infinite domain* (e.g. the set of integers), and (2) many properties of interest require *quantification* over the array elements. For instance, the property "array a is sorted in ascending order" is

*Corresponding author

Authors' addresses: Chih-Duo Hong, National Chengchi University, Taipei, Taiwan, chihduo@nccu.edu.tw; Anthony W. Lin, University of Kaiserslautern-Landau, Kaiserslautern, Germany and MPI-SWS, Kaiserslautern, Germany, awlin@mpi-sws.org.

ACM 2475-1421/2024/1-ART22
https://doi.org/10.1145/3632864

expressed by a universally quantified formula stating that each element in the array a is no smaller than its preceding elements. For this reason, in order to verify the correctness of an array system, it is often necessary to reason about quantified formulae, which is generally undecidable over arrays (cf. [Bradley and Manna 1998; Kroening and Strichman 2016]).

Owing to the undecidability of quantified array theories, existing techniques for handling array systems employ a mixture of SMT, model checking, synthesis, and/or abstraction. Most effort in developing SMT over arrays concentrates on providing more support of universal quantification [Bradley et al. 2006; Cimatti et al. 2021; Ge and De Moura 2009; Habermehl et al. 2008; Mann et al. 2022]. To lift SMT over arrays to the verification of array systems, one may exploit model checking techniques like IC3/PDR (e.g. [Cimatti et al. 2016; Gurfinkel et al. 2018; Komuravelli et al. 2015]), backward reachability (e.g. [Abdulla et al. 2009; Ranise and Ghilardi 2010]), synthesis (e.g. SyGuS [Fedyukovich et al. 2019]), eager abstraction (e.g. [Mann et al. 2022; McMillan 2018]), interpolation (e.g. [Ghilardi et al. 2021; Hoenicke and Schindler 2018; McMillan 2008]), or predicate abstraction [Flanagan and Qadeer 2002; Lahiri and Bryant 2004b, 2007], among others. It should be remarked that, besides a handful of work (e.g. [Padon et al. 2017, 2021]), most existing work on array systems verification concerns *safety verification*, with liveness still posing a significant challenge.

Contributions. This paper presents a new method for reasoning about array systems. We demonstrate its efficacy for verifying safety and liveness of challenging examples from array programs and distributed protocols, including Dijkstra's self-stabilizing protocol and the Chang-Roberts leader election protocol. The method has two main ingredients. Firstly, we provide a new *predicate abstraction* (using *indexed predicates*) that overapproximates an array system as a *string rewriting system*. String rewriting systems can be construed as array systems, whose array elements range over a *finite domain* (a.k.a. alphabet). The second ingredient answers why we use string rewriting systems as abstractions: this subclass of array systems has been more amenable to solutions than the general case of arrays. In fact, several powerful verification methods over string rewriting systems (for *both* safety and liveness, among others) have been developed in the last decade or so, which include methods in the framework *regular model checking* [Abdulla 2012; Bouajjani et al. 2004, 2000; Lin and Rümmer 2022]. This framework relies on the decidable first-order theory over words with prefix-of-relation, regular constraints, and the equal-length predicate (a.k.a. the universal automatic structure [Benedikt et al. 2003; Blumensath and Gradel 2000]). The theory of universal automatic structure can be construed as a kind of array theory, whose elements range over a finite domain (e.g. bitvectors). In stark contrast to decidable array theories like [Bradley et al. 2006; Habermehl et al. 2008], however, the theory has no restrictions on the use of quantifiers, but restricts the manner in which the indices are related. The set of solutions definable by such formulae are precisely those that are captured by synchronized automata running on tuples of words (a.k.a. *regular relations*).

We now provide some details of our method. We use a variant of *First-Order Linear Temporal Logic (FO-LTL)* [Abadi 1989; Hodkinson et al. 2000] restricted to a quantified array theory, in order to specify both an array system and properties to be verified. This formalism may also be regarded as a variant of *indexed LTL* [Clarke et al. 1986; German and Sistla 1992] with atomic propositions replaced by expressions from the array theory. Our predicate abstraction reduces this FO-LTL model checking problem over array systems to a verification problem over string rewriting systems that can be modeled in the framework of regular model checking. We discuss next our notion of predicate abstraction by means of indexed predicates. Indexed predicates have been used in previous work (e.g. in [Flanagan and Qadeer 2002] for invariant generation and in [Lahiri and Bryant 2007] for abstracting into a finite system), but never in the context of computing another infinite-state system that is more amenable to analysis. To this end, the right logic instantiating the indexed

Table 1. Comparison between our approach and the classical methods. The first column is the setting of the classical (indexed) predicate abstraction (e.g. [Flanagan and Qadeer 2002; Jhala et al. 2018; Lahiri and Bryant 2007]). The second column provides our setting of indexed predicate abstraction (Section 4.2). The last column is our approximation of indexed predicate abstraction using regular languages (Section 4.4), which we refer to as *regular abstraction*.

	Predicate Abstraction	Indexed Predicate Abstraction	Regular Abstraction
Abstract Set	finite set of bitvectors	infinite set of words over bitvectors	finite-state automaton
Abstract Relation	finite relation over bitvectors	infinite relation over words	finite-state transducer
Abstract System	finite Boolean program	infinite-state transition system	regular transition system

predicates has to be devised that suits our abstraction as string rewriting systems. In our case, each *indexed predicate* is an atomic formula (a.k.a. atom) with a designated variable i, e.g., $a[i] \leq a[n]$. Given indexed predicates $\langle P_1, \ldots, P_m \rangle$, interpretations of an array formula ϕ are mapped into a word w over the alphabet $\{0,1\}^m$ of m-bitvectors. This word w summarizes the truth values of the predicates over all positions in the arrays interpreting ϕ.

Although we have chosen to abstract away the actual array values by means of such indexed predicates, our abstraction maintains the concrete values of array sizes and index variables in the abstract domain. This strategy often allows us to compute a precise enough abstraction based on a small set of predicates. Given an interpretation σ of an array formula (as numbers and arrays), the set $\alpha(\sigma)$ of abstract values corresponding to σ (as tuples of numbers and words over a finite alphabet) is computable. We propose next to overapproximate sets and relations in the abstract domain with regular sets and relations. We show that abstractions of quantified array formulae and array systems can be suitably approximated using regular language as a symbolic representation. This yields our notion of *regular abstractions* for array formulae and systems. Figure 1 compares this approach with the classical predicate abstraction.

We show that regular abstractions are sufficiently precise for verification purposes when the quantified array formulae used in the FO-LTL specifications belong to the so-called *Singly-Indexed Array Logic (SIA)* [Habermehl et al. 2008]. To evaluate our approach, we consider non-trivial safety and liveness properties for several case studies, including selection and merge sort algorithms, Dijkstra's self-stabilizing algorithm [Dijkstra 1982], and the Chang-Roberts leader election algorithm [Chang and Roberts 1979]. We report promising experimental results for these case studies using off-the-shelf regular model checkers as backend inference engines.

2 AN ILLUSTRATIVE EXAMPLE

Dijkstra's self-stabilizing algorithm [Dijkstra 1982] assumes a ring of $n \geq 2$ processes with identifiers $1, \ldots, n$ and n local variables $x_1, \ldots, x_n$ over $\{0, \ldots, k-1\}$, where n and k are finite but unbounded parameters satisfying $n \leq k$. For $i \in \{1, \ldots, n\}$, Process i is *privileged* if (1) $i = 1$ and $x_1 = x_n$, or (2) $i > 1$ and $x_i \neq x_{i-1}$. At each time step, a privileged process, say Process i, is scheduled, and the corresponding variable x_i is updated by setting (1) $x_i := x_i + 1 \bmod k$ when $i = 1$, or (2) $x_i := x_{i-1}$ when $i > 1$. Process i thus loses its privilege after the update. Dijkstra's self-stabilizing algorithm can be construed as a kind of token-passing algorithm, whereby a process holds a token if and only if it is privileged. The algorithm is self-stabilizing in the sense that eventually the system will contain a single token forever.

Let us identify each variable x_i with an array[1] value $a[i]$, and define a formula

$$\mathsf{Priv}(i) := (i = 1 \wedge a[i] = a[n]) \vee (i > 1 \wedge i \leq n \wedge a[i] \neq a[i-1])$$

indicating that Process i is privileged. One interesting property of Dijkstra's algorithm is that Process 1 will be privileged infinitely often, regardless of what the initial state is and how the privileged processes are scheduled. To prove this, we may use an index variable pid to represent the scheduled process, namely, $pid = 1$ means that Process 1 is scheduled. Then the property "Process 1 is privileged infinitely often" can be expressed in LTL as $\mathbf{GF}\,\mathsf{Priv}(1)$. We would like to prove that this property holds for Dijkstra's algorithm under the scheduling assumption $\mathbf{G}\,\mathsf{Priv}(pid)$, namely, under the assumption that the scheduler always selects a privileged process.

Inspired by the definition of $\mathsf{Priv}(i)$, we may abstract the local state of each Process i using predicates $\langle a[i] = a[n], a[i] = a[i-1]\rangle$. The induced abstract state space is then $\mathbb{N} \times \mathbb{N} \times \Sigma^*$ with $\Sigma := \{00, 10, 01, 11\}$. Each abstract state (pid, n, w) comprises the valuations of pid and n, and a word w over the alphabet Σ. The i-th letter of w essentially corresponds to the *predicate abstraction* of a at position i. For example, $w[2] = 01$ means that a satisfies $\neg(a[2] = a[n]) \wedge (a[2] = a[1])$. The abstraction function α then maps a concrete state, say $s := (1, 3, [1, 2, 1])$, to a set of abstract states, say $\alpha(s) = \{(1, 3, [10, 00, 10], (1, 3, [11, 00, 10])\}$. Observe that we may naively overapproximate the abstraction of (the concrete states satisfying) $\mathsf{Priv}(1)$ by $F := \mathbb{N} \times \mathbb{N} \times (10 + 11) \cdot \Sigma^*$. Here, F is an overapproximation in the sense that $\alpha(s) \not\subseteq F$ implies $s \not\models \mathsf{Priv}(1)$, whereas $\alpha(s) \subseteq F$ implies nothing. (For example, given $s := (1, 2, [1, 2])$, we can deduce $s \not\models \mathsf{Priv}(1)$ by observing that $\alpha(s) = \{(1, 2, [00, 10]), (1, 2, [01, 10])\} \not\subseteq F$.) Similarly, the abstraction of $\mathsf{Priv}(pid)$ may be overapproximated by $S := (\{1\} \times \mathbb{N} \times (10 + 11) \cdot \Sigma^*) \uplus \bigcup_{i \geq 2} \left(\{i\} \times \mathbb{N} \times \Sigma^{i-1} \cdot (00 + 10) \cdot \Sigma^*\right)$. Thus, $\mathbf{GF}\,\mathsf{Priv}(1)$ holds for Dijkstra's algorithm under the scheduling assumption $\mathbf{G}\,\mathsf{Priv}(pid)$ if there does not exist a maximal abstract path $\pi \in S^*$ satisfying $\mathbf{FG}\,(S \setminus F)$, namely, π consists of states in S and eventually stays away from F forever.

In this manner, we can reduce verification problems of an array manipulating system to those of a string manipulating system, for which effective techniques and highly optimized tools exist (e.g. [Abdulla et al. 2012; Chen et al. 2017; Fiedor et al. 2017; Klarlund and Møller 2001; Klarlund et al. 2002; Lin and Rümmer 2016; Schuppan and Biere 2006]). Unfortunately, the approximations S and F given above are too rough to verify the desired property, in the sense that they will produce spurious counterexample paths after the reduction. We shall later show that an appropriate choice of predicates and approximations will allow us to verify this property.

3 PRELIMINARIES

Notation. We adopt the standard notation of many-sorted logic (see e.g. [van Dalen 1994]), and consistently use $\mathcal{V}$ to denote a set of sorted first-order variables. We use $\Sigma(\mathcal{V})$ to denote the set of sort-consistent interpretations of $\mathcal{V}$. Given two interpretations $\sigma_1 \in \Sigma(\mathcal{V}_1)$ and $\sigma_2 \in \Sigma(\mathcal{V}_2)$ with $\mathcal{V}_1 \cap \mathcal{V}_2 = \emptyset$, we use $\sigma_1 \cdot \sigma_2$ to denote the interpretation $\sigma \in \Sigma(\mathcal{V}_1 \uplus \mathcal{V}_2)$ such that $\sigma(v) = \sigma_1(v)$ for $v \in \mathcal{V}_1$ and $\sigma(u) = \sigma_2(u)$ for $u \in \mathcal{V}_2$. We write v' for the primed version of variable v, and define $\mathcal{V}' := \{v' : v \in \mathcal{V}\}$. Given $\sigma \in \Sigma(\mathcal{V})$, we use σ' to denote the interpretation $\{v' \mapsto \sigma(v) : v \in \mathcal{V}\}$ of $\mathcal{V}'$. Given a formula ϕ, we use $\phi[t/x]$ to denote the formula obtained by substituting t for all free occurrences of x in ϕ. Finally, for ϕ defined over variables $\mathcal{V}$, we let $[\![\phi]\!] := \{\sigma \in \Sigma(\mathcal{V}) : \sigma \models \phi\}$, and write $\phi \equiv \psi$ when $[\![\phi]\!] = [\![\psi]\!]$.

Indexed array logic. An array theory typically combines two one-sorted theories: an *index theory*, which is used to express relations between indices, and an *element theory*, which is used to

[1] In this paper, array indices always start from 1. We use $a[i]$ to denote the i-th element of array a, and use $|a|$ to denote the size of a. We often use $[a_1, \ldots, a_n]$ to represent an unnamed array a with $|a| = n$ and $a[i] = a_i$ for $i \in \{1, \ldots, n\}$.

express properties of the array entries. The *indexed array logic* is a fragment of quantified array theory that (i) uses Difference Arithmetic as the index theory, and (ii) allows quantification only over index variables. To make our expositions concrete, we shall focus on an instantiation of indexed array logic that uses Difference Arithmetic as the element theory. (However, we note that our abstraction techniques can be directly applied to any indexed array logic with a decidable quantifier-free fragment, which allows for a rich set of element theories such as Linear Integer Arithmetic (cf. [Kroening and Strichman 2016]).)

Throughout this paper, we shall fix an indexed array logic $\mathcal{T}$ with syntax given by

$$I ::= I + n \mid i \mid n \mid |a| \qquad \text{(index terms)}$$
$$E ::= E + n \mid v \mid n \mid a[I] \qquad \text{(data terms)}$$
$$F ::= I \bowtie I \mid E \bowtie E \mid \neg F \mid F \vee F \mid F \wedge F \mid \exists i.\, F \mid \forall i.\, F \qquad \text{(formulae)}$$

where $\bowtie \in \{=, \neq, \leq, <, \geq, >\}$, n is a concrete integer, v is a data variable, i is an index variable, a is an array variable, $|a|$ represents the array length of a, and $a[I]$ represents the array element of a stored at position I. A formula in form of $I \bowtie I$ or $E \bowtie E$ is called an *atomic formula*, or an *atom* for short. We extend the logic with logical connectives such as $\Rightarrow$ (implies) and $\Leftrightarrow$ (if and only if) in the usual way. An interpretation σ of a formula in the indexed array logic is intuitive: σ maps each integer/index variable to an integer, and each array variable a to a function $f_a : \mathbb{Z} \to \mathbb{Z}$. The notion of satisfaction of a formula with respect to this interpretation is now standard. Although arrays are infinite, it is easy to confine an array a to a finite array by focusing on the index range $\{1, \ldots, |a|\}$. For example, a formula $\forall i.\, \phi(i, a)$ can simply be rewritten to $\forall i.\, 1 \leq i \wedge i \leq |a| \Rightarrow \phi(i, a)$.

First-order array system. A *first-order array system* $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$ is a triple where $\mathcal{V}$ is a finite set of first-order variables, and ϕ_I and ϕ_T are array formulae over variables $\mathcal{V}$ and $\mathcal{V} \uplus \mathcal{V}'$, respectively. The variables in $\mathcal{V}$ are grouped into *state variables* and *system parameters*. A state variable has either an index sort, an element sort, or an array sort. A system parameter is a special index variable whose value is determined in an initial state and immutable during system execution. For each array variable $a \in \mathcal{V}$, we identify a system parameter in $\mathcal{V}$ with the array size $|a|$. This effectively stipulates that the array size is not changeable after the array is allocated.

The semantics of an array system is determined by the array logic $\mathcal{T}$ and interpretations for $\mathcal{V}$. A *state* of an array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$ is a sort-consistent interpretation $s \in \Sigma(\mathcal{V})$ for the variables in $\mathcal{V}$. The formula ϕ_I specifies the set of initial states. The formula ϕ_T specifies the system's evolution by relating the current variables $\mathcal{V}$ to their updated counterparts $\mathcal{V}'$. A (finite or infinite) sequence $\pi := s_0 \cdots s_n$ of states with $n \in \mathbb{N} \cup \{\omega\}$ is a *path* if $s_0 \models \phi_I$ and $s_{i-1} \cdot s'_i \models \phi_T$ for $i \in \{1, \ldots, n\}$. Since system parameters are immutable, π is a legitimate path only if $s_{i-1}(v) = s_i(v)$ for all system parameters $v \in \mathcal{V}$ and each $i \in \{1, \ldots, n\}$.

A guarded modeling language. We describe a simple guarded command language for specifying transitions of array systems. While this language is less expressive than the full-fledged array logic, it is more concise and understandable, and is already expressive enough to capture all examples considered in this paper. A guarded command is in form of *guard* $\triangleright$ *update*. A command is *enabled* if its guard is satisfied. At each time step, the system nondeterministically selects an enabled command and updates the current state accordingly. If no command is enabled, the system halts. More precisely, a guarded command is in form of $\phi \triangleright \beta_1, \ldots, \beta_m$, where ϕ is a (possibly quantified) array formula, and each β_i is an assignment in one of the two forms:

- $x := I$ or $x := *$, where x is an index variable, I is an index term, and $*$ is a special symbol;
- $a[I] := E$ or $a[I] := *$, where I is an index term, E is a data term, and $*$ is a special symbol.

The assignments $\beta_1, \ldots, \beta_m$ are executed in parallel. The symbol "$*$" stands for a nondeterministic value, which ranges over the index domain when the left-hand side is x, and ranges over the

data domain when the left-hand side is $a[I]$. For simplicity, assignments to data variables are not directly supported by our modeling language. This however is not an essential restriction, since data variables can be formally translated to array elements stored at dedicated positions, thereby allowing assignments as array elements.

Example 3.1. The transitions of Dijkstra's algorithm in the illustrative example may be given as

$$pid = 1 \wedge a[pid] = a[n] \wedge a[pid] < k-1 \rhd a[pid] := a[pid]+1,\ pid := *$$
$$pid = 1 \wedge a[pid] = a[n] \wedge a[pid] = k-1 \rhd a[pid] := 0,\ pid := *$$
$$pid > 1 \wedge pid \le n \wedge a[pid] \neq a[pid-1] \rhd a[pid] := a[pid-1],\ pid := *$$

The array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$ of Dijkstra's algorithm is specified by

$$\mathcal{V} := \{pid, a, n, k\}$$
$$\phi_I := 2 \le n \wedge n \le k \wedge (\forall i.\ 1 \le i \wedge i \le n \Rightarrow 0 \le a[i] \wedge a[i] < k)$$
$$\phi_T := \psi_1 \vee \psi_2 \vee \psi_3$$

where pid is an index variable, a is an array variable, n, k are system parameters, and $|a|$ is identified with n. The transition formula ϕ_T is defined by three formulae ψ_1, ψ_2, and ψ_3, which are in turn derived from the three guarded commands listed above. For example, ψ_1 can be formally derived from the first command as $pid = 1 \wedge a[pid] = a[n] \wedge a[pid] < k-1 \wedge a'[pid] = a[pid]+1 \wedge 1 \le pid' \wedge pid' \le n \wedge (\forall i.\ 1 \le i \wedge i \le n \wedge i \neq pid \Rightarrow a'[i] = a[i])$. □

4 REGULAR LANGUAGES AS ABSTRACTIONS

In this section, we define a novel predicate abstraction of array formulae and array systems using indexed predicates. Since each array will be mapped to a word (over a finite alphabet) in the abstraction, we shall use regular languages to represent sets of words, and use regular relations to represent relations over words. We start by recalling a logical framework of regular relations.

4.1 A First-Order Framework of Regular Relations

Let $\Sigma_m := \{0,1\}^m$ denote the set of bitvectors of length m. For $m \ge 1$, we define a two-sorted structure[2] $\mathfrak{S}_m := (\Sigma_m^*, \mathbb{N}, succ, pred, \cdot[\cdot], |\cdot|, \Delta_1, \ldots, \Delta_m)$, where $\Sigma_m^* := \{v_1 \cdots v_n : n \ge 0$ and $v_i \in \Sigma_m$ for each $i\}$ are words over bitvectors; $succ, pred : \mathbb{N} \to \mathbb{N}$ are defined by $succ(n) := n+1$ and $pred(n) := \max\{0, n-1\}$, respectively; $|\cdot| : \Sigma_m^* \to \mathbb{N}$ is word length (i.e. $|w|$ is the length of w); $\cdot[\cdot] : \Sigma_m^* \times \mathbb{N} \to \Sigma_m$ offers letter-level access for words, i.e., $w[i]$ is the ith letter of w for $1 \le i \le |w|$; each $\Delta_k \subseteq \mathbb{N} \times \Sigma_m^*$ offers bit-level access for words, such that $(i, w) \in \Delta_k$ if and only if $1 \le i \le |w|$ and the kth bit of $w[i]$ is 1. We shall use $n+k$ and $n-k$ as syntactic sugar for $succ^k(n)$ and $pred^k(n)$, respectively.

Given two words $w_1 \in \Sigma_{m_1}^*$ and $w_2 \in \Sigma_{m_2}^*$, we define the *convolution* $w_1 \odot w_2$ of w_1 and w_2 as the word $w \in \Sigma_{m_1+m_2}^*$ such that $|w| = \max\{|w_1|, |w_2|\}$ and

$$w[i] = \begin{cases} w_1[i] :: w_2[i], & 1 \le i \le \min\{|w_1|, |w_2|\}; \\ 0^{m_1} :: w_2[i], & |w_1| < i \le |w_2|; \\ w_1[i] :: 0^{m_2}, & |w_2| < i \le |w_1|, \end{cases} \tag{1}$$

where $u :: v$ denotes the concatenation of the bitvectors u and v. Intuitively, $w_1 \odot w_2$ is obtained by juxtaposing w_1 and w_2 (in a left-aligned manner), and padding the shorter word with enough

[2]We note that this structure is equally expressive in first-order logic to the so-called *universal automatic structures* [Benedikt et al. 2003; Blumensath and Grädel 2004; Colcombet and Löding 2007].

0s. For example, for two words $\mathtt{01}, \mathtt{0001}$ over Σ_1, $\mathtt{01} \odot \mathtt{0001}$ is the word $(0,0)(1,0)(0,0)(0,1)$ over Σ_2. We lift $\odot$ to languages by defining $L_1 \odot L_2 := \{w_1 \odot w_2 : w_1 \in L_1,\ w_2 \in L_2\}$.

We say that a relation is *regular* if its language representation is regular under unary encoding. Formally, suppose w.l.o.g. that $R \subseteq \mathbb{N}^r \times (\Sigma_m^*)^l$. We define the *language representation* of R as $\mathcal{L}(R) := \{w_1 \odot \cdots \odot w_{r+l} \in \Sigma_{r+l\cdot m}^* : (w_1, \ldots, w_{r+l}) \in R\}$ by identifying each $n \in \mathbb{N}$ with $1^n \in \Sigma_1^*$. The following result states that a relation is regular if and only if it is first-order definable in $\mathfrak{S}_m$, see e.g., [Benedikt et al. 2003; Blumensath and Grädel 2004; Colcombet and Löding 2007].

Proposition 4.1. *Given a first-order formula ϕ defined over $\mathfrak{S}_m$, we can compute a finite automaton recognizing $\mathcal{L}(\llbracket\phi\rrbracket)$. Conversely, given a finite automaton recognizing $\mathcal{L} \subseteq N_1 \odot \cdots \odot N_r \odot L_1 \odot \cdots \odot L_l$, where $N_i \subseteq 1^*$ and $L_i \subseteq \Sigma_m^*$ for each i, we can compute a first-order formula ϕ over $\mathfrak{S}_m$ such that $\mathcal{L} = \mathcal{L}(\llbracket\phi\rrbracket)$.*

4.2 Abstraction of Array Formulae

We use indexed predicates to express constraints on arrays.

Definition 4.2 (Indexed predicate). An *indexed predicate* is an atom (in the indexed array logic) containing a designated index variable $\mathtt{i}$. A set of indexed predicates $\mathcal{P} := \langle P_1, \ldots, P_m\rangle$ is said to be defined over variables $\mathcal{V}$ if each predicate $P_k \in \mathcal{P}$ is defined over variables $\mathcal{V} \uplus \{\mathtt{i}\}$.

For convenience, we shall fix a set of variables $\mathcal{V}$ and a set of indexed predicates $\mathcal{P} := \langle P_1, \ldots, P_m\rangle$ over variables $\mathcal{V}$ throughout the rest of this section. For each bitvector $v := b_1 \cdots b_m \in \Sigma_m$ comprising m bits, we define a quantifier-free array formula $\phi_v^{\mathcal{P}}$ over variables $\mathcal{V} \uplus \{\mathtt{i}\}$ as follows:

$$\phi_v^{\mathcal{P}} := \bigwedge\nolimits_{1\le k\le m} \tilde{P_k}, \text{ where } \tilde{P_k} = \begin{cases} P_k, & b_k = 1; \\ \neg P_k, & b_k = 0. \end{cases} \tag{2}$$

Formula $\phi_v^{\mathcal{P}}$ effectively regards each bit b_k of the bitvector v as the truth value of predicate P_k at the parametric position $\mathtt{i}$. We furthermore use a word over bitvectors to encode the truth values of the predicates ranging over an interval of positions: for each word $w \in \Sigma_m^*$, we define an array formula

$$\Phi_w^{\mathcal{P}} := \bigwedge\nolimits_{1\le i\le |w|} \phi_{w[i]}^{\mathcal{P}}[i/\mathtt{i}]. \tag{3}$$

For example, if $\mathcal{P} = \langle x[\mathtt{i}] = x[n], x[\mathtt{i}] \neq x[\mathtt{i}-1]\rangle$ and $w = [\mathtt{10}, \mathtt{01}]$, then $\Phi_w^{\mathcal{P}}$ is $x[1] = x[n] \wedge \neg(x[1] \neq x[0]) \wedge \neg(x[2] = x[n]) \wedge x[2] \neq x[1]$. By construction, for $\sigma \in \Sigma(\mathcal{V})$, $\sigma \models \Phi_w^{\mathcal{P}}$ holds if and only if for $i \in \{1, \ldots, |w|\}$ and $k \in \{1, \ldots, m\}$, the truth value of $\sigma \models P_k[i/\mathtt{i}]$ coincides with the kth bit of the bitvector $w[i]$.

Fix an array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$. Let $\mathcal{V}_{int}$ denote the set of parameters *and* index variables in $\mathcal{V}$, and $\mathcal{V}_{par}$ denote the set of parameters in $\mathcal{V}$. Let $r := |\mathcal{V}_{int}|$ and $m := |\mathcal{P}|$. We shall identify an interpretation $\sigma \in \Sigma(\mathcal{V}_{int})$ with an r-tuple $(\sigma(x_1), \ldots, \sigma(x_r)) \in \mathbb{N}^r$ where x_i denotes the ith variable in $\mathcal{V}_{int}$. We define the *concrete domain* of the system as $\Sigma(\mathcal{V})$, and the *abstract domain* as $\mathbb{N}^r \times \Sigma_m^*$. Each abstract state $(t, w) \in \mathbb{N}^r \times \Sigma_m^*$ comprises a tuple $t \in \mathbb{N}^r$ representing an interpretation of $\mathcal{V}_{int}$, and a word $w \in \Sigma_m^*$ encoding the valuations of $\mathcal{P}$ at positions $1, \ldots, |w|$.

Given a set $S_C \subseteq \Sigma(\mathcal{V})$ of concrete states, we define

$$\alpha^{\mathcal{P}}(S_C) := \{(s_{int}, w) \in \mathbb{N}^r \times \Sigma_m^* : \text{there exists } s \in S_C \text{ s.t. } |w| = \max_{x\in\mathcal{V}_{par}} s(x) \text{ and } s \models \Phi_w^{\mathcal{P}}\}.$$

Here, s_{int} denotes the interpretation obtained by restricting s to $\mathcal{V}_{int}$. Given a set $S_A \subseteq \mathbb{N}^r \times \Sigma_m^*$ of abstract states, we define

$$\gamma^{\mathcal{P}}(S_A) := \{s \in \Sigma(\mathcal{V}) : \text{for all } w \in \Sigma_m^* \text{ s.t. } |w| = \max_{x\in\mathcal{V}_{par}} s(x),\ s \models \Phi_w^{\mathcal{P}} \text{ implies } (s_{int}, w) \in S_A\}.$$

Notice that $\alpha^{\mathcal{P}}(S_C)$ and $\gamma^{\mathcal{P}}(S_A)$ are computable when S_C and S_A are finite. To simplify the notation, we may omit the superscript $\mathcal{P}$ from $\alpha^{\mathcal{P}}$ and $\gamma^{\mathcal{P}}$ when it is clear from the context. Also, for a single state s, we shall write $\alpha(s)$ and $\gamma(s)$ instead of $\alpha(\{s\})$ and $\gamma(\{s\})$ when there is no danger of ambiguity. Finally, for a formula ϕ, we shall write $\alpha(\phi)$ and $\gamma(\phi)$ instead of $\alpha([\![\phi]\!])$ and $\gamma([\![\phi]\!])$.

Example 4.3. Recall the illustrative example in Section 2. With indexed predicates $\mathcal{P} := \langle a[\mathtt{i}] = a[n], a[\mathtt{i}] = a[\mathtt{i}-1]\rangle$, we can define α and γ as

$$\alpha(S_C) := \{(pid, n, w) \in \mathbb{N}^2 \times \Sigma_2^* : \text{there exists } (pid, n, a) \in S_C \text{ s.t. } |w| = n \text{ and } (pid, n, a) \models \Phi_w^{\mathcal{P}}\};$$

$$\gamma(S_A) := \{(pid, n, a) \in \mathbb{N}^2 \times \mathbb{Z}^\omega : \text{for all } w \in \Sigma_2^n,\ (pid, n, a) \models \Phi_w^{\mathcal{P}} \text{ implies } (pid, n, w) \in S_A\}.$$

For instance, given a concrete state $s := (1, 3, [1, 2, 1])$, we can compute $\alpha(s) = \{(1, 3, [\mathtt{10}, \mathtt{00}, \mathtt{10}]), (1, 3, [\mathtt{11}, \mathtt{00}, \mathtt{10}])\}$ and $\gamma(\alpha(s)) = \{(1, 3, a) : a \in \mathbb{Z}^\omega,\ a[1] = a[3] \neq a[2]\}$. □

Lemma 4.4. *For a countable collection $\{S_i : i \in I\}$ of concrete state sets, it holds that $\alpha(\bigcup_{i\in I} S_i) = \bigcup_{i\in I} \alpha(S_i)$ and $\alpha(\bigcap_{i\in I} S_i) \subseteq \bigcap_{i\in I} \alpha(S_i)$.*

Proof. Note that α distributes over union by definition, i.e., $\alpha(S) = \bigcup_{s\in S} \alpha(s)$. Also, note that $s \in \bigcap_{i\in I} S_i$ implies $\alpha(s) \subseteq \bigcap_{i\in I} \alpha(S_i)$. Thus, $\alpha(\bigcap_{i\in I} S_i) = \bigcup\{\alpha(s) : s \in \bigcap_{i\in I} S_i\} \subseteq \bigcap_{i\in I} \alpha(S_i)$. □

Lemma 4.5. *For any abstract state set S_A, it holds that $\gamma(S_A) = \{s : \alpha(s) \subseteq S_A\}$.*

Proof. It suffices to fix a concrete state s and show that $s \in \gamma(S_A) \iff \alpha(s) \subseteq S_A$. For the "$\Rightarrow$" direction, assume that $s \in \gamma(S_A)$, and let (s_{int}, w) be an abstract state in $\alpha(s)$. Then we have $s \models \Phi_w^{\mathcal{P}}$ by the definition of α, which implies that $(s_{int}, w) \in S_A$ by the definition of γ. Since (s_{int}, w) is arbitrary, we conclude that $\alpha(s) \subseteq S_A$. For the "$\Leftarrow$" direction, assume that $\alpha(s) \subseteq S_A$. Let $W_s := \{w \in \Sigma_m^* : |w| = \max_{x\in\mathcal{V}_{par}} s(x)\}$. By the definition of α, S_A contains all abstract states (s_{int}, w) such that $w \in W_s$ and $s \models \Phi_w^{\mathcal{P}}$. However, this implies that for any $w \in W_s$, if $s \models \Phi_w^{\mathcal{P}}$ holds, then $(s_{int}, w) \in S_A$ also holds. It follows that $s \in \gamma(S_A)$ by the definition of γ. □

The following result is a consequence of Lemma 4.4, Lemma 4.5, and Proposition 7 of [Cousot and Cousot 1977]. We provide a proof here for the sake of completeness.

Proposition 4.6. *For any concrete state set S_C and abstract state set S_A, it holds that*

$$\alpha(S_C) \subseteq S_A \iff S_C \subseteq \gamma(S_A).$$

Proof. For the "$\Rightarrow$" direction, assume that $\alpha(S_C) \subseteq S_A$. Since α is monotonic, i.e., $E \subseteq F$ implies $\alpha(E) \subseteq \alpha(F)$, we have $\alpha(s) \subseteq S_A$ for every $s \in S_C$. By Lemma 4.5, we have $\gamma(S_A) = \{s : \alpha(s) \subseteq S_A\}$. It then follows that $S_C \subseteq \gamma(S_A)$. For the "$\Leftarrow$" direction, assume that $S_C \subseteq \gamma(S_A)$. Again by Lemma 4.5, we have $\gamma(S_A) = \{s : \alpha(s) \subseteq S_A\}$. The assumption $S_C \subseteq \gamma(S_A)$ thus implies that $\alpha(s) \subseteq S_A$ for every $s \in S_C$. This in turn implies that $\alpha(S_C) \subseteq S_A$ by Lemma 4.4. □

4.3 Abstract Safety and Liveness Analysis

For an array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$, applying predicate abstraction for safety and liveness verification involves performing reachability analysis for the abstract system of $\mathfrak{T}$. In each step of the analysis, we concretize the current set of reachable abstract states via the concretization operation γ, apply the concrete next-step function

$$N_C(S_C) := \{t \in \Sigma(\mathcal{V}) : \text{there exists } s \in S_C \text{ such that } s{\cdot}t' \models \phi_T\} \tag{4}$$

in the concrete state space, and map the result back to the abstract state space via the abstraction operation α. We can view this process as performing state exploration in an abstract transition system with a set $\alpha(\phi_I)$ of initial states and an abstract next-step function

$$N_A(S_A) := \alpha(N_C(\gamma(S_A))). \tag{5}$$

Proposition 4.6, together with the fact that α, γ, and N_C are monotonic, establishes the soundness of abstract reachability analysis using N_A (cf. [Cousot and Cousot 1977]). For example, let $\Pi(I, T)$ denote the set of paths starting from states in I through transition relation T, and let $st(\pi)$ denote the set of states in path π. Given a set $Init$ of initial states and a set Bad of bad states, one can show (e.g. by induction on the length of a counterexample path) that the safety property

$$\forall \pi \in \Pi(Init, N_C).\ st(\pi) \cap Bad = \emptyset \tag{6}$$

holds in the concrete domain if $\forall \pi \in \Pi(\alpha(Init), N_A).\ st(\pi) \cap \alpha(Bad) = \emptyset$ holds in the abstract domain. Similarly, given a set Fin of final states, the liveness property

$$\forall \pi \in \Pi(Init, N_C).\ st(\pi) \cap Fin \neq \emptyset \tag{7}$$

holds in the concrete domain if $\forall \pi \in \Pi(\alpha(Init), N_A).\ st(\pi) \cap \alpha(Fin^{\mathsf{c}})^{\mathsf{c}} \neq \emptyset$ holds in the abstract domain,where S^{c} denotes the complement of the concrete (resp. abstract) state set S with respect to the concrete (resp. abstract) state space. These facts allow us to reduce the verification of concrete safety/liveness properties to that of their counterparts in the abstract domain.

4.4 Approximation with Regular Languages

Since indexed array logic is undecidable, the abstraction $\alpha(\phi)$ of an indexed array formula ϕ is generally not computable. Therefore, we propose to overapproximate these formulae in the abstract domain using regular languages. As before, fix a set $\mathcal{V}$ of variables and a set $\mathcal{P}$ of indexed predicates over $\mathcal{V}$, and let $r := |\mathcal{V}_{int}|$ and $m := |\mathcal{P}|$. Then $\mathcal{V}$ and $\mathcal{P}$ induce an abstract domain $\mathbb{N}^r \times \Sigma_m^*$. Recall that we regard a relation R as regular when R's language representation $\mathcal{L}(R)$ is regular.

Definition 4.7 (Regular abstraction for state formulae). Let ϕ be an indexed array formula over variables $\mathcal{V}$. Then a tuple $(R, \mathcal{P})$ is a *regular abstraction* of ϕ if $R \subseteq \mathbb{N}^r \times \Sigma_m^*$ is a regular relation such that $\alpha^{\mathcal{P}}(\phi) \subseteq R$.

For $\mathcal{P} := \langle P_1, \ldots, P_m \rangle$ over $\mathcal{V}$, we use $\mathcal{P} * \mathcal{P}' := \langle P_1, \ldots, P_m, P_{m+1}, \ldots, P_{2m} \rangle$ to denote the set of predicates over $\mathcal{V} \uplus \mathcal{V}'$ such that for $k \in \{1, \ldots, m\}$, P_{m+k} is obtained from P_k by replacing v with v' for each variable $v \in \mathcal{V}$. Recall that $s{\cdot}t$ denotes the union of interpretations s and t over disjoint variables, and that $u \odot v$ denotes the convolution of words u and v.

Definition 4.8 (Regular abstraction for transition formulae). Let ϕ be an indexed array formula over variables $\mathcal{V} \uplus \mathcal{V}'$. Then a tuple $(R, \mathcal{P})$ is a *regular abstraction* of ϕ if $R \subseteq (\mathbb{N}^r \times \Sigma_m^*) \times (\mathbb{N}^r \times \Sigma_m^*)$ is a regular relation such that $\alpha^{\mathcal{P} * \mathcal{P}'}(\phi) \subseteq \tau(R)$, where τ is an isomorphic mapping defined by $\tau((s, u), (t, v)) = (s{\cdot}t, u \odot v)$.

A regular abstraction $(R, \mathcal{P})$ of a transition formula induces a next-step function N_R, defined by $N_R(S_A) := \{t : \text{there exists } s \in S_A \text{ such that } (s, t) \in R\}$, over the abstract state sets. It can be shown that N_R provides an *abstract interpretation* [Cousot and Cousot 1977] of the concrete array system.

Theorem 4.9. *Let $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$ be an array system and $\mathcal{P}$ be a set of indexed predicates over $\mathcal{V}$. If $(R, \mathcal{P})$ is a regular abstraction of ϕ_T, then N_R provides an abstract interpretation of $\mathfrak{T}$.*

Proof. To show that N_R is an abstract interpretation, we need to check that (1) N_R is monotonic; (2) N_R is null-preserving, namely, $N_R(\emptyset) = \emptyset$; (3) N_R simulates N_C w.r.t. a simulation relation defined by α, namely, $\alpha(N_C(S_C)) \subseteq N_R(\alpha(S_C))$. The first two conditions directly follow from the definition of regular abstraction, so it suffices to check the last condition. Suppose first that $S_C = \emptyset$.

Since $N_C(\emptyset) = \alpha(\emptyset) = N_R(\emptyset) = \emptyset$, the condition holds trivially. Now suppose that $S_C \neq \emptyset$ and consider an arbitrary state $s \in S_C$. Observe that

$$\begin{aligned}\alpha(N_C(s)) &= \alpha(\{t \in \Sigma(\mathcal{V}) : s{\cdot}t' \models \phi_T\}) \\ &= \{(t_{int}, v) : s{\cdot}t' \models \phi_T,\ |v| = \max_{x \in \mathcal{V}_{par}} t(x),\ t \models \Phi_v^{\mathcal{P}}\} \\ &\subseteq N_R(\{(s_{int}, u) : |u| = \max_{x \in \mathcal{V}_{par}} s(x),\ s \models \Phi_u^{\mathcal{P}}\}) \\ &= N_R(\alpha(s)),\end{aligned}$$

where the inclusion in the third line follows from the definition of regular abstraction and the stipulation that parameters are immutable. Note that N_C, N_R, and α are all distributive over union. Since s is arbitrary in S_C, we have $\alpha(N_C(S_C)) = \bigcup_{s \in S_C} \alpha(N_C(s)) \subseteq \bigcup_{s \in S_C} N_R(\alpha(s)) = N_R(\alpha(S_C))$. □

Finally, a regular abstraction can be computed by composing smaller regular abstractions. This fact is an immediate consequence of Lemma 4.4.

Proposition 4.10. *Let $\phi_1, \ldots, \phi_n$ be indexed array formulae, and $(R_i, \mathcal{P})$ be a regular abstraction of ϕ_i for $i \in \{1, \ldots, n\}$. Then $(\bigcup_{i=1}^n R_i, \mathcal{P})$ is a regular abstraction of $\bigvee_{i=1}^n \phi_i$, and $(\bigcap_{i=1}^n R_i, \mathcal{P})$ is a regular abstraction of $\bigwedge_{i=1}^n \phi_i$.*

5 COMPUTATION OF REGULAR ABSTRACTIONS

We have proposed to overapproximate indexed predicate abstractions with regular abstractions. However, it remains unclear how to find such approximations that are sufficiently precise for a verification task. In this section, we address this issue by providing an automated procedure to compute regular abstractions for a fragment of array logic called *singly indexed array formulae*. In a nutshell, give such a formula $\phi := \delta x_1 \cdots \delta x_n.\, \varphi$, we compute an overapproximation of ϕ by replacing the matrix φ of ϕ with a quantifier-free formula $\tilde{\varphi}$ such that $\varphi \Rightarrow \tilde{\varphi}$ is valid. We shall choose $\tilde{\varphi}$ in such a way that the resulting formula $\delta x_1 \cdots \delta x_n.\, \tilde{\varphi}$, called an *abstraction formula*, can be faithfully encoded as a first-order formula over an automatic structure. This encoding is equally expressive to regular languages, and can be conveniently manipulated using tools such as Mona [Klarlund and Møller 2001; Klarlund et al. 2002] and Gaston [Fiedor et al. 2017] for abstract analysis.

5.1 Singly Indexed Array Formulae

An atomic indexed array formula is a *data expression* if it contains an array variable; otherwise it is an *index expression*. A *singly indexed array (SIA)* formula is an indexed array formula in which every data expression contains *at most* one quantified index variable. The satisfiability problem of SIA formulae is already undecidable when the element theory is instantiated to Difference Arithmetic [Habermehl et al. 2008, Lemma 5]. Note that an SIA formula can be syntactically rewritten to a logically equivalent SIA formula where every data expression has *exactly* one quantified index variable — hence the name "singly indexed". For example, a data expression $a[1] \neq b[n]$ in an SIA formula can be replaced with $\exists i.\, i = 1 \wedge a[i] \neq b[n]$ to obtain an equivalent SIA formula. We shall assume an SIA formula to have this form when we are computing a regular abstraction for the formula.

5.2 A Constraint-Based Abstraction Procedure

We need a couple more definitions before we introduce our abstraction method. As before, fix a set $\mathcal{V}$ of variables and a set $\mathcal{P} := \langle P_1, \ldots, P_m \rangle$ of indexed predicates over $\mathcal{V}$. A formula over $\mathcal{V}$ is *expressible in* $\mathcal{P}$ if each atom of the formula is either an index expression or an expression

of form $P[t/\mathtt{i}]$ or $\neg P[t/\mathtt{i}]$, where t is an index term and P is a predicate in $\mathcal{P}$. For a formula ϕ expressible in $\mathcal{P}$, we shall use $\kappa^{\mathcal{P}}(\phi)$ to denote the formula obtained by replacing each atom $P_k[t/\mathtt{i}]$ (resp. $\neg P_k[t/\mathtt{i}]$) in ϕ with $\psi \Rightarrow \Delta_k(t, w)$ (resp. $\psi \Rightarrow \neg\Delta_k(t, w)$), where ψ encodes the boundary check for $P_k[t/\mathtt{i}]$. For example, suppose that $\phi = \forall i.\, a[i] \leq b[p]$ and that $P_1 = a[\mathtt{i}] > b[p]$ is a predicate in $\mathcal{P}$. Then $\kappa^{\mathcal{P}}(\phi) = \forall i.\, (1 \leq i \wedge i \leq |a| \wedge 1 \leq p \wedge p \leq |b| \Rightarrow \neg\Delta_1(i, w))$. Note that $\kappa^{\mathcal{P}}(\phi)$ is defined over structure $\mathfrak{S}_m$ and variables $\mathcal{V}_{int} \uplus \{w\}$. Specifically, the free variables of $\kappa^{\mathcal{P}}(\phi)$ comprise the index variables and parameters of ϕ, as well as a fresh word variable w over domain Σ_m^*. We may simply write $\kappa^{\mathcal{P}}(\phi)$ as $\kappa(\phi)$ when $\mathcal{P}$ is clear from the context.

Now we are ready to describe our abstraction method. Consider a singly indexed formula ϕ over $\mathcal{V}$. Suppose w.l.o.g. that the matrix of ϕ is given in disjunctive normal form $\bigvee_j (g_j \wedge h_j)$, where each g_j is a conjunction of index expressions, and each h_j is a conjunction of data expressions. Our procedure computes a regular abstraction for ϕ in two steps:

Step 1: Replace each h_j in ϕ with a formula $\mathrm{cstr}(h_j)$, where $\mathrm{cstr}(\psi)$ denotes the conjunction of disjunctive clauses defined by

$$\mathrm{cstr}(\psi) := \bigwedge \{C \subseteq A(\psi, \mathcal{P}) : \psi \Rightarrow C \text{ is valid in } \mathcal{T}\}, \tag{8}$$

and $A(\psi, \mathcal{P}) := \{P_k[t/\mathtt{i}], \neg P_k[t/\mathtt{i}] : P_k \in \mathcal{P}$, and t is an index term of ψ or $\mathcal{P}\}$. Intuitively, $\mathrm{cstr}(\psi)$ attempts to express in $\mathcal{P}$ the necessary conditions for a concrete state to satisfy ψ. Note that $A(\psi, \mathcal{P})$ is finite, and hence $\mathrm{cstr}(\psi)$ is computable by our assumption on $\mathcal{T}$.

Step 2: Let ϕ^* denote the formula obtained by replacing each h_j with $\mathrm{cstr}(h_j)$. This formula ϕ^*, which we shall refer to as the *abstraction formula* of ϕ, is expressible in $\mathcal{P}$. We then define $\phi^{\mathcal{P}} = \phi^{\mathcal{P}}(\sigma, w) := \kappa^{\mathcal{P}}(\phi^*)$, and use $(\llbracket \phi^{\mathcal{P}} \rrbracket, \mathcal{P})$ as a regular abstraction for ϕ.

Note that when computing $\mathrm{cstr}(\psi)$, we can exclude a clause C if $\kappa^{\mathcal{P}}(C)$ is valid in $\mathfrak{S}_m$ (e.g. when $\{f, \neg f\} \subseteq C$ for some atom f), since such a clause imposes no restrictions on the abstract states. Furthermore, it suffices to consider the *minimal* clauses C satisfying $\psi \Rightarrow C$. Thus, we can reduce the computation of $\mathrm{cstr}(\psi)$ to enumerating the *minimal unsatisfiable cores (MUCs)* of $\psi \wedge \bigwedge A(\psi, \mathcal{P})$, as negating such a core essentially leads to a minimal feasible clause of $\mathrm{cstr}(\psi)$. There are rich tool supports for MUC enumeration in the literature, see e.g. [Bendík and Černá 2020; Bendík et al. 2018; Liffiton et al. 2016]. In practice, it is often possible to compute $\mathrm{cstr}(\psi)$ incrementally and obtain a precise enough abstraction without including all MUCs in the abstraction formula. We shall discuss these optimizations in Section 8.

Example 5.1. As an illustration, let us use our abstraction method to prove that there always exists at least one privileged process in Dijkstra's self-stabilizing algorithm (cf. Example 3.1). Consider the singly indexed array formula

$$\phi := \forall i.\, (i = 1 \wedge i < n \wedge a[i] \neq a[n]) \vee (i > 1 \wedge i \leq n \wedge a[i] = a[i-1]) \vee (i > n \wedge n \geq 2),$$

which expresses that no process is privileged in a system containing $n \geq 2$ processes. We shall compute a regular abstraction of ϕ using indexed predicates $\mathcal{P} := \langle a[\mathtt{i}] = a[n],\, a[\mathtt{i}] = a[\mathtt{i}-1] \rangle$, which comprises the atomic formulae of ϕ. The first step computes the abstraction formula ϕ^* as

$$\forall i.\, (i = 1 \wedge i < n \wedge \mathrm{cstr}(a[i] \neq a[n])) \vee (i > 1 \wedge i \leq n \wedge \mathrm{cstr}(a[i] = a[i-1])) \vee (i > n \wedge n \geq 2).$$

The nontrivial minimal feasible clauses for $\mathrm{cstr}(a[i] \neq a[n])$ are $\{a[n] = a[n]\}$, $\{a[i] \neq a[n]\}$, and $\{a[i] \neq a[i-1], a[i-1] \neq a[n]\}$. (We call a clause C trivial if $\kappa(C)$ is valid in $\mathfrak{S}_m$.) Similarly, the nontrivial minimal feasible clauses for $\mathrm{cstr}(a[i] = a[i-1])$ are $\{a[n] = a[n]\}$, $\{a[i] = a[i-1]\}$, $\{a[i] = a[n], a[i-1] \neq a[n]\}$, and $\{a[i] \neq a[n], a[i-1] = a[n]\}$. From these clauses, we then compute a regular abstraction of ϕ as $\phi^{\mathcal{P}} = \phi^{\mathcal{P}}(n, w) := \kappa(\phi^*) = \forall i.\, (i = 1 \wedge i < n \wedge \kappa(\mathrm{cstr}(a[i] \neq$

$a[n]))) \vee (i > 1 \wedge i \leq n \wedge \kappa(\mathrm{cstr}(a[i] = a[i-1]))) \vee (i > n \wedge n \geq 2)$. For example,

$$\begin{aligned}\kappa(\mathrm{cstr}(a[i] \neq a[n])) &= \kappa(a[n] = a[n] \wedge a[i] \neq a[n] \wedge (a[i] \neq a[i-1] \vee a[i-1] \neq a[n])) \\ &= (1 \leq n \wedge n \leq n \Rightarrow \Delta_1(n, w)) \wedge (1 \leq i \wedge i \leq n \Rightarrow \neg\Delta_1(i, w)) \wedge \\ &\quad ((1 \leq i \wedge i \leq n \Rightarrow \neg\Delta_2(i, w)) \vee (1 \leq i - 1 \wedge i - 1 \leq n \Rightarrow \neg\Delta_1(i-1, w))).\end{aligned}$$

We can effectively check that $\phi^{\mathcal{P}}$ is unsatisfiable in $\mathfrak{S}_2$ by Proposition 4.1. Since regular abstractions are overapproximations, this implies that ϕ is unsatisfiable in $\mathcal{T}$. In this way, with the provided indexed predicates, our tool can prove that ϕ is unsat in a second, while solvers like Z3 and cvc5 fail to handle the formula (i.e. both of them output "unknown" for checking satisfiability of ϕ.) □

Transition formulae. Given a singly indexed transition formula ϕ over variables $\mathcal{V} \uplus \mathcal{V}'$, we can compute a regular abstraction for ϕ by first extending the predicates $\mathcal{P}$ to $\mathcal{P} \uplus \mathcal{P}'$, where $\mathcal{P}'$ is obtained from $\mathcal{P}$ by replacing each $x \in \mathcal{V}$ with x'. We then compute an abstract transition formula $\phi^{\mathcal{P}}((\sigma, w), (\sigma', w'))$ analogously to how we compute an abstract state formula $\phi^{\mathcal{P}}(\sigma, w)$. However, if we only consider transition formulae induced by our guarded command language, we can compute an abstract transition formula directly from a command as follows. Define $\mathrm{copy\_except}(x) := \bigwedge_{z \in \mathcal{V} \setminus \{x\}}(z' = z)$. Consider w.l.o.g. a guarded command $\psi \triangleright \beta$, where β is in form of either $x := e$ or $a[t] := e$. For $x := e$, the corresponding transition formula is

$$\psi \wedge \mathrm{copy\_except}(x) \wedge x' = e. \tag{9}$$

For $a[t] := e$, the corresponding formula is

$$\psi \wedge \mathrm{copy\_except}(a) \wedge a' = a\{t \leftarrow e\}, \tag{10}$$

where $a\{t \leftarrow e\}$ is a notation from the theory of extensional arrays [McCarthy 1993], denoting the array obtained by assigning e to $a[t]$. (We may set e to a fresh variable when e is the nondeterministic symbol "$*$".) Note that the abstraction formulae of (9) and (10) are computable. Indeed, for an array formula τ and a clause C, the validity check of $(\tau \wedge a' = a\{t \leftarrow e\}) \Rightarrow C$ can be compiled into a decidable array theory supported by most SMT solvers (cf. [Kroening and Strichman 2016]).

5.3 Correctness of the Abstraction Procedure

Let $\phi := \delta x_1 \cdots \delta x_n.\, \varphi$ be a singly indexed array formula, where φ is quantifier-free. Recall that, to compute a regular abstraction of ϕ, our procedure first computes an abstraction formula $\phi^* = \delta x_1 \cdots \delta x_n.\, \tilde{\varphi}$ by replacing φ with a quantifier-free formula $\tilde{\varphi}$ such that (1) $\tilde{\varphi}$ is in negation normal form (NNF), (2) $\tilde{\varphi}$ is expressible in $\mathcal{P}$, and (3) $\varphi \Rightarrow \tilde{\varphi}$ holds in $\mathcal{T}$. Our procedure then outputs the formula $\phi^{\mathcal{P}} = \kappa(\phi^*)$, where $\kappa(\cdot)$ replaces each atom $P_k[t/\mathtt{i}]$ (resp. $\neg P_k[t/\mathtt{i}]$) in the formula with $\psi \Rightarrow \Delta_k(\sigma, w)$ (resp. $\psi \Rightarrow \neg\Delta_k(t, w)$), using ψ to encode the boundary check of $P_k[t/\mathtt{i}]$. Now we show that this procedure is sound, that is, $(\llbracket \phi^{\mathcal{P}} \rrbracket, \mathcal{P})$ is indeed a regular abstraction of ϕ.

Lemma 5.2. *Let $\mathcal{P}$ be a set of indexed predicates, and ϕ be a singly indexed array formula that is expressible in $\mathcal{P}$. If ϕ is satisfiable, then $\kappa(\mathsf{NNF}(\phi))$ is also satisfiable, where $\mathsf{NNF}(\phi)$ denotes the negation normal form of ϕ.*

Proof. Fix a set of predicates $\mathcal{P} = \langle P_1, \ldots, P_m \rangle$. For an SIA formula ϕ over $\mathcal{V}$, suppose that σ is a solution of ϕ. Then σ uniquely determines a word $w = w(\sigma) \in \Sigma_m^*$ such that $\sigma \models \Phi_w^{\mathcal{P}}$ and $|w| = \max\{\sigma(x) : x \text{ is a parameter in } \mathcal{V}\}$. Particularly, for each $i \in \{1, \ldots, |w|\}$ and $k \in \{1, \ldots, m\}$, $w \models \Delta_k(i, w)$ in $\mathfrak{S}_m$ if and only if $\sigma \models P_k[i/\mathtt{i}]$ in $\mathcal{T}$ (cf. Section 4.2). Now we argue that (σ_{int}, w) is a solution of $\kappa(\phi)$, where σ_{int} denotes the restriction of σ to index variables and parameters in $\mathcal{V}$. We shall prove this by induction on the quantifier rank of ϕ, denoted by $rank(\phi)$.

For the base case, we show that $\sigma_{int}, w \models \kappa(\phi)$ when $rank(\phi) = 0$ (i.e. ϕ is quantifier-free). Since ϕ is in NNF, we can assume w.l.o.g. that $\phi = g \wedge h$ and $\kappa(\phi) = g \wedge \kappa(h)$ by writing ϕ in DNF. Here, g

is a conjunction of index expressions, and $h = \bigwedge_j h_j$ is a conjunction of data expressions. Assume to the contrary that $\sigma \models \phi$ but $\sigma_{int}, w \not\models \kappa(\phi)$. Since $\sigma_{int} \models g$, we have $\sigma_{int}, w \models \neg\kappa(h_j)$ for some j. Since h_j is expressible in $\mathcal{P}$, suppose w.l.o.g. that $h_j = P_k[t/\mathtt{i}]$ for some $P_k \in \mathcal{P}$ and index term t. Then $\sigma_{int}, w \models 1 \leq t \wedge t \leq |w| \wedge \neg\Delta_k(t, w)$ holds by the definition of κ. But this implies $\sigma \models \neg h_j$ (and hence $\sigma \not\models \phi$) by the definition of w, leading to a contradiction. It follows that $\sigma_{int}, w \models \kappa(\phi)$, i.e., the hypothesis is true when $rank(\phi) = 0$.

Now suppose that $rank(\phi) = n+1$, and that the hypothesis holds for formulae of rank equal to n. If $\sigma \models \exists i.\, \psi$, we have $\sigma \models \psi[p/i]$ for some p. Then $\sigma_{int}, w \models \kappa(\psi[p/i])$ by the induction hypothesis, which implies that $\sigma_{int}, w \models \kappa(\exists i.\, \psi)$. Similarly, if $\sigma \models \forall i.\, \psi$, we have $\sigma \models \psi[p/i]$ for all p. Then $\sigma_{int}, w \models \kappa(\psi[p/i])$ for all p by the induction hypothesis, which implies that $\sigma_{int}, w \models \kappa(\forall i.\, \psi)$. The statement therefore holds for formulae of all ranks by induction. □

Theorem 5.3 (Correctness). *Let $\mathcal{P}$ be a set of indexed predicates, and ϕ be a singly indexed array formula. Then $[\![\phi^{\mathcal{P}}]\!]$ is regular and $\alpha(\phi) \subseteq [\![\phi^{\mathcal{P}}]\!]$. Namely, $\phi^{\mathcal{P}}$ yields a regular abstraction of ϕ.*

Proof. $[\![\phi^{\mathcal{P}}]\!]$ is regular by Proposition 4.1. Since α is monotonic and $\phi \implies \phi^*$, it suffices to show that $\alpha(\phi^*) \subseteq [\![\phi^{\mathcal{P}}]\!]$. Consider an arbitrary abstract state $(\sigma, w) \in \alpha(\phi^*)$, with σ being an interpretation of index variables and system parameters $x_1, \ldots, x_r$ in $\mathcal{V}$. Define $\Psi_{\sigma,w} := (\bigwedge_{i=1}^{r} x_i = \sigma(x_i)) \wedge \Phi^{\mathcal{P}}_w$. Then $\phi^* \wedge \Psi_{\sigma,w}$ is satisfiable by the definition of α. Since $\phi^* \wedge \Psi_{\sigma,w}$ is in NNF and expressible in $\mathcal{P}$, $\kappa(\phi^* \wedge \Psi_{\sigma,w})$ is satisfiable by Lemma 5.2. However, observe that (σ, w) is a solution of $\kappa(\phi^* \wedge \Psi_{\sigma,w})$ when $\kappa(\phi^* \wedge \Psi_{\sigma,w})$ is satisfiable, as the formula only constrains its free word variable on the first $|w|$ letters. Therefore, we have $(\sigma, w) \in [\![\kappa(\phi^* \wedge \Psi_{\sigma,w})]\!] = [\![\kappa(\phi^*) \wedge \kappa(\Psi_{\sigma,w})]\!] \subseteq [\![\kappa(\phi^*)]\!] = [\![\phi^{\mathcal{P}}]\!]$. This allows us to conclude that $\alpha(\phi^*) \subseteq [\![\phi^{\mathcal{P}}]\!]$, and therefore $\alpha(\phi) \subseteq [\![\phi^{\mathcal{P}}]\!]$. □

5.4 A Closure Property

Consider a finite set Φ of singly indexed array formulae. Note that we can always select a set of indexed predicates that expresses all formulae in Φ. In such cases, if Φ is closed under Boolean operations, then the abstractions our procedure computes for Φ are also closed under Boolean operations, as is indicated by the following proposition.

Proposition 5.4. *For indexed predicates $\mathcal{P}$ and singly indexed array formulae ϕ and ψ, it holds that $(\phi \vee \psi)^{\mathcal{P}} \equiv \phi^{\mathcal{P}} \vee \psi^{\mathcal{P}}$. Furthermore, we have $(\phi \wedge \psi)^{\mathcal{P}} \equiv \phi^{\mathcal{P}} \wedge \psi^{\mathcal{P}}$ and $(\neg\phi)^{\mathcal{P}} \equiv \neg\phi^{\mathcal{P}} \wedge \mathsf{true}^{\mathcal{P}}$[3] when ϕ and ψ are expressible in $\mathcal{P}$.*

Proof. Suppose w.l.o.g. that $\phi = \delta x_1 \cdots \delta x_p.\, \bigvee_{i \in I}(g_i \wedge h_i)$ and $\psi = \delta y_1 \cdots \delta y_q.\, \bigvee_{j \in J}(g_j \wedge h_j)$, where (i) $I \cap J = \emptyset$, and (ii) for each $k \in I \uplus J$, g_k is a conjunction of index expressions, and h_k is a conjunction of data expressions. Then we have

$$(\phi \vee \psi)^{\mathcal{P}} \equiv \delta x_1 \cdots \delta x_p\, \delta y_1 \cdots \delta y_q.\, \bigvee_{k \in I \uplus J} g_k \wedge \kappa(\mathsf{cstr}(h_k)) \equiv \phi^{\mathcal{P}} \vee \psi^{\mathcal{P}}.$$

Furthermore, notice that

$$(\phi \wedge \psi)^{\mathcal{P}} \equiv \delta x_1 \cdots \delta x_p\, \delta y_1 \cdots \delta y_q.\, \bigvee_{i \in I} \bigvee_{j \in J} g_i \wedge g_j \wedge \kappa(\mathsf{cstr}(h_i \wedge h_j))$$
$$\phi^{\mathcal{P}} \wedge \psi^{\mathcal{P}} \equiv \delta x_1 \cdots \delta x_p\, \delta y_1 \cdots \delta y_q.\, \bigvee_{i \in I} \bigvee_{j \in J} g_i \wedge g_j \wedge \kappa(\mathsf{cstr}(h_i)) \wedge \kappa(\mathsf{cstr}(h_j))$$

Hence, it suffices to show that $\kappa(\mathsf{cstr}(h_i \wedge h_j)) \Leftrightarrow \kappa(\mathsf{cstr}(h_i)) \wedge \kappa(\mathsf{cstr}(h_j))$ holds in $\mathfrak{S}_m$. For the "$\Rightarrow$" direction, note that a feasible clause for $\mathsf{cstr}(h_i)$ *or* $\mathsf{cstr}(h_j)$ is also feasible for $\mathsf{cstr}(h_i \wedge h_j)$. Thus, a solution of $\kappa(\mathsf{cstr}(h_i \wedge h_j))$ is also a solution of $\kappa(\mathsf{cstr}(h_i)) \wedge \phi(\mathsf{cstr}(h_j))$. For the "$\Leftarrow$" direction, suppose that C is a feasible clause for $\mathsf{cstr}(h_i \wedge h_j)$, and thus $\kappa(\mathsf{cstr}(h_i \wedge h_j)) \Rightarrow \kappa(C)$ holds in $\mathfrak{S}_m$.

[3] One may regard $[\![\mathsf{true}^{\mathcal{P}}]\!]$ as the abstract state space induced by $\mathcal{P}$ in our regular abstraction framework.

Since h_i, h_j are expressible in $\mathcal{P}$, $\neg h_j \vee C$ and $\neg h_i \vee C$ are feasible clauses for $\mathrm{cstr}(h_i)$ and $\mathrm{cstr}(h_j)$, respectively. Similarly, each atom of h_i and h_j induces a singleton feasible clause for $\mathrm{cstr}(h_i)$ and $\mathrm{cstr}(h_j)$, respectively. Hence, $\kappa(\mathrm{cstr}(h_i) \wedge \mathrm{cstr}(h_j)) \Rightarrow \kappa(h_i \wedge h_j \wedge (\neg h_i \vee C) \wedge (\neg h_j \vee C)) \Rightarrow \kappa(h_i \wedge h_j \wedge C) \Rightarrow \kappa(C)$ holds in $\mathfrak{S}_m$. Thus, a solution of $\kappa(\mathrm{cstr}(h_i)) \wedge \kappa(\mathrm{cstr}(h_j))$ is also a solution of $\kappa(\mathrm{cstr}(h_i \wedge h_j))$. This completes the proof for both directions, leading to $(\phi \wedge \psi)^{\mathcal{P}} \equiv \phi^{\mathcal{P}} \wedge \psi^{\mathcal{P}}$.

Finally, note that $(\neg\phi)^{\mathcal{P}} \wedge \phi^{\mathcal{P}} \equiv (\neg\phi \wedge \phi)^{\mathcal{P}} \equiv \mathsf{false}^{\mathcal{P}} \equiv \mathsf{false}$, and $(\neg\phi)^{\mathcal{P}} \vee \phi^{\mathcal{P}} \equiv (\neg\phi \vee \phi)^{\mathcal{P}} \equiv \mathsf{true}^{\mathcal{P}}$. This allows us to deduce that $(\neg\phi)^{\mathcal{P}} \equiv \neg\phi^{\mathcal{P}} \wedge \mathsf{true}^{\mathcal{P}}$. □

Proposition 5.4 makes it possible to compute precise regular abstractions of complex formulae by composing abstractions of simple formulae. For example, we may compute the regular abstraction of (10) as $\phi^{\mathcal{P}} = \psi^{\mathcal{P}} \wedge (\mathsf{copy\_except}(a) \wedge a' = a\{t \leftarrow e\})^{\mathcal{P}}$. The two regular abstractions on the right-hand side can then be reused for computing abstractions of other formulae. Even if ϕ is not expressible in $\mathcal{P}$, we can still compute $\phi^{\mathcal{P}}$ using composition since regular abstractions are closed under conjunction (Proposition 4.10), but at the price of potentially losing precision.

We conclude this section with the following observation.

Proposition 5.5. *Let $\mathcal{P}$ be a set of indexed predicates, and ϕ be a singly indexed array formula that is expressible in $\mathcal{P}$. Then it holds that $\gamma(\phi^{\mathcal{P}}) = \gamma(\alpha(\phi))$.*

Proof. Since $\alpha(\phi) \subseteq [\![\phi^{\mathcal{P}}]\!]$ by Theorem 5.3, we have $\gamma(\alpha(\phi)) \subseteq \gamma(\phi^{\mathcal{P}})$. Suppose to the contrary that there exists $s \in \gamma(\phi^{\mathcal{P}}) \setminus \gamma(\alpha(\phi))$. Since $s \in \gamma(\phi^{\mathcal{P}})$, we have $\alpha(s) \subseteq [\![\phi^{\mathcal{P}}]\!]$ by Proposition 4.6. Similarly, since $s \notin \gamma(\alpha(\phi))$, we have $\alpha(s) \not\subseteq \alpha(\phi)$ by Proposition 4.6. The latter fact implies that $s \not\models \phi$, that is, $s \models \neg\phi$. Thus, $\alpha(s) \subseteq \alpha(\neg\phi) \subseteq [\![(\neg\phi)^{\mathcal{P}}]\!] = [\![\mathsf{true}^{\mathcal{P}}]\!] \setminus [\![\phi^{\mathcal{P}}]\!]$ by Theorem 5.3 and Proposition 5.4. It follows that $\alpha(s) \cap [\![\phi^{\mathcal{P}}]\!] = \emptyset$, a contradiction. This completes the proof. □

6 VERIFICATION OF TEMPORAL ARRAY PROPERTIES

In this section, we discuss how to verify linear-time array system properties by combining our abstraction techniques with regular model checking. We shall first introduce the notion of temporal array properties and abstractable specifications. We then present two verification methods for typical safety and liveness array properties. Finally, we describe a generic technique to verify a syntactic fragment of array properties called the index-bounded monodic properties.

6.1 Temporal Array Property

To express the temporal properties of an array system, we provide a specification language combining the indexed array logic with LTL. This can be seen as a restriction of FO-LTL [Abadi 1989; Hodkinson et al. 2000] to the indexed array logic. For ease of exposition, we shall only consider the "globally" and "eventually" connectives in the sequel. Our approach however can be extended to handle other standard connectives such as "next" and "until" in a straightforward manner.

A *temporal array property* is a formula ϕ constrained by

$$\phi ::= \psi \mid \neg\phi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists i.\, \phi \mid \forall i.\, \phi \mid \mathbf{G}\,\phi \mid \mathbf{F}\,\phi$$

where ψ is an indexed array formula, i is an index variable, and $\mathbf{G}$ and $\mathbf{F}$ are the standard "globally" and "eventually" temporal connectives, respectively, in LTL. As usual, we extend the logic with logical operators such as $\Rightarrow$ (implies) and $\Leftrightarrow$ (if and only if). The semantics of a temporal array property is standard (see e.g. [Hodkinson et al. 2000]). Intuitively, temporal connectives offer means of relating two states at different time points in a path, whereas the first-order quantifiers allow one to relate different array positions. For example, both $\mathbf{G}\,(\forall i.\, 1 \le i \wedge i \le |a| \Rightarrow a[i] = 0)$ and $\forall i.\, (1 \le i \wedge i \le |a| \Rightarrow \mathbf{G}\, a[i] = 0)$ assert that the array a has only 0 as elements throughout the path. We say that a temporal array property ϕ holds for an array system if the system does not have a path that satisfies $\neg\phi$.

6.2 Abstraction of Array System Specifications

Definition 6.1 (Abstractable specification). An array formula is *abstractable* if it is of form $\exists \overline{i}.\, \phi$ for some (possibly quantified) SIA formula ϕ. An array system is *abstractable* if it is specified with abstractable array formulae. A temporal array property ϕ is *abstractable* if it is constrained by

$$\phi ::= \psi \mid \phi \vee \phi \mid \phi \wedge \phi \mid \exists i.\, \phi \mid \mathbf{G}\,\phi \mid \mathbf{F}\,\phi$$

for (possibly quantified) SIA formulae ψ.

Abstractable formulae slightly generalize SIA formulae by allowing multiple existentially indexed/quantified variables. In fact, a formula is abstractable if and only if it can be transformed into an SIA formula using *Skolemization*. Thus, an abstractable first-order array system $\mathfrak{T}$ can be formally transformed into a set of SIA formulae. More precisely, suppose that $\phi := \exists \overline{i}.\, \psi$ is an abstractable array formula over variables $\mathcal{V}$. Define an SIA formula $\psi^\star := \psi[\overline{c}/\overline{i}]$ over variables $\mathcal{V} \uplus \{\overline{c}\}$, where $\overline{c}$ are fresh system parameters. We can then add the new parameters $\overline{c}$ to the vocabulary of $\mathfrak{T}$ and substitute $\psi^\star$ for ϕ in the specification. Clearly, a temporal array property holds for $\mathfrak{T}$ if and only if the property holds for the Skolemized version of $\mathfrak{T}$.

An abstractable temporal array property can furthermore be converted to an equisatisfiable temporal array property wherein all maximal non-temporal subformulae are singly indexed. The conversion is essentially the same as Skolemization, except that here we replace an indexed variable quantified inside a **G** operator with a fresh index variable, and replace an indexed variable that is *only* quantified inside an **F** operator with a fresh system parameter. The maximal non-temporal subformulae of the obtained temporal array property are all singly indexed, meaning that their semantics can be overapproximated by regular abstractions. In other words, an abstractable temporal array property can be abstracted to a propositional temporal formula with atoms being regular languages. This fact makes it possible to leverage existing verification techniques in regular model checking to perform abstract analysis for abstractable specifications and properties.

6.3 Safety Verification

A *safety array property* is a temporal array property in form of $\forall \overline{i}.\, (\psi_1 \Rightarrow \mathbf{G}\,\psi_2)$. Fix a safety array property ϕ and an array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$. Suppose that both $\mathfrak{T}$ and $\neg\phi \equiv \exists \overline{i}.\, (\psi_1 \wedge \mathbf{F}\,\neg\psi_2)$ are abstractable. Define $\psi_1^\star := \psi_1[\overline{c}/\overline{i}]$ and $\psi_2^\star := \psi_2[\overline{c}/\overline{i}]$ with fresh system parameters $\overline{c}$. Moreover, define a transition system $\mathfrak{T}^\star := (\mathcal{V}^\star, \phi_I^\star, \phi_T, \phi_B)$ with $\mathcal{V}^\star := \mathcal{V} \uplus \{\overline{c}\}$, $\phi_I^\star := \phi_I \wedge \psi_1^\star$, and $\phi_B := \neg\psi_2^\star$. It is clear that $\mathfrak{T}^\star$ is abstractable. Given a set $\mathcal{P}$ of indexed predicates over $\mathcal{V}^\star$, we can compute regular abstractions $(I, \mathcal{P})$, $(T, \mathcal{P})$, and $(B, \mathcal{P})$ of $\phi_I^\star$, ϕ_T, and ϕ_B, respectively, as described in Section 5. We then obtain a regular safety property (I, T, B) that holds iff $T^*(I) \cap B = \emptyset$, where T^* denotes the transitive closure of T. It is clear that (I, T, B) holds only if $\mathfrak{T}$ satisfies the safety array property ϕ. With Proposition 4.1, we can summarize this result as follows.

Theorem 6.2. *Let $\mathfrak{T}$ be an array system, and ϕ be a safety array property. Suppose that $\mathfrak{T}$ and $\neg\phi$ are abstractable. Given a set of indexed predicates, we can effectively compute finite automata $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{B}$ such that ϕ holds for $\mathfrak{T}$ if the safety property $(\mathcal{I}, \mathcal{T}, \mathcal{B})$ holds.*

6.4 Liveness Verification

A *liveness array property* is a temporal array property in form of $\forall \overline{i}.\, (\psi_1 \Rightarrow \mathbf{F}\,\psi_2)$. (We pick this "eventuality" property for simplicity and sufficiency for our benchmarks, although our technique can easily be adapted to other liveness properties such as recurrence). Fix a liveness array property ϕ and an array transition system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$. Suppose that both $\mathfrak{T}$ and $\neg\phi \equiv \exists \overline{i}.\, (\psi_1 \wedge \mathbf{G}\,\neg\psi_2)$ are abstractable. Define $\psi_1^\star := \psi_1[\overline{c}/\overline{i}]$ and $\psi_2^\star := \psi_2[\overline{c}/\overline{i}]$ with fresh system parameters $\overline{c}$, and an array transition system $\mathfrak{T}^\star := (\mathcal{V}^\star, \phi_I^\star, \phi_T, \phi_F)$, where $\mathcal{V}^\star := \mathcal{V} \uplus \{\overline{c}\}$, $\phi_I^\star := \phi_I \wedge \psi_1^\star$, and

$\phi_F := \psi_2^\star$. It is clear that $\mathfrak{T}^\star$ is abstractable. Given a set $\mathcal{P}$ of indexed predicates over $\mathcal{V}^\star$, we compute regular abstractions $(I, \mathcal{P})$, $(T, \mathcal{P})$, and $(E, \mathcal{P})$ for $\phi_I^\star$, ϕ_T, and $\neg\phi_F$, respectively. Now, following the reduction described in Section 4.3, we let $F := \Sigma_m^* \setminus E$ and define a regular liveness property (I, T, F) that holds if and only if $st(\pi) \cap F \neq \emptyset$ for every path $\pi \in \Pi(I, T)$. It is not hard to see that the abstract transition system $\langle \Sigma_m^*, I, T\rangle$ preserves *infinite* counterexample paths from $\mathfrak{T}$. That is, if there exists an infinite concrete path $s_0\, s_1 \cdots$ such that $s_0 \models \psi_I^\star$ and $s_i \not\models \phi_F$ for all $i \geq 0$, then there exists an infinite abstract path $t_0\, t_1 \cdots$ such that $t_0 \in I$ and $t_i \notin F$ for all $i \geq 0$.

We say that a transition system is *$\neg\phi$-progressing* if every maximal system path satisfying $\neg\phi$ is an infinite path. Observe that when $\mathfrak{T}$ is $\neg\phi$-progressing, (I, T, F) holds only if $\mathfrak{T}$ satisfies the liveness array property ϕ. We hence have the following result.

Theorem 6.3. *Let $\mathfrak{T}$ be an array system, and ϕ be a liveness array property. Suppose that $\mathfrak{T}$ and $\neg\phi$ are abstractable, and that $\mathfrak{T}$ is $\neg\phi$-progressing. Then given a set of indexed predicates, we can effectively compute finite automata $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{F}$ such that ϕ holds for $\mathfrak{T}$ if the liveness property $(\mathcal{I}, \mathcal{T}, \mathcal{F})$ holds.*

Progress verification. Theorem 6.2 and 6.3 exploit the same technique that overapproximates the target array property with a counterpart property of the abstract system. Generally, this technique works for any temporal array property with an abstractable negated formula, as long as a false property remains false after new transitions are introduced to the system. For a safety property, this requirement is met automatically since extensions of a counterexample path are still counterexample paths. For a liveness array property ϕ, this requires the array system to be $\neg\phi$-progressing, i.e., the system does not have a finite maximal counterexample path.

We may formulate and verify the progress condition as a safety array property as follows. Given an array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$ and a liveness array property $\phi := \forall \bar{i}.\, (\psi_1 \Rightarrow \mathbf{F}\psi_2)$, we aim to check if $\mathfrak{T}$ is $\neg\phi$-progressing. As before, define $\psi_1^\star := \psi_1[\bar{c}/\bar{i}]$ and $\psi_2^\star := \psi_2[\bar{c}/\bar{i}]$ with fresh system parameters $\bar{c}$. We then specify an array system $\mathfrak{T}^* := (\mathcal{V}^\star, \phi_I^\star, \phi_T^\star)$ with $\mathcal{V}^\star := \mathcal{V} \uplus \{\bar{c}\}$, $\phi_I^\star := \phi_I \wedge \psi_1^\star$, and $\phi_T^\star := \phi_T \wedge \neg\psi_2^\star$. It is easy to see that the system $\mathfrak{T}^*$ is safe w.r.t. the safety array property $\phi^* := \forall \bar{i}.\, \mathbf{G}\, \exists \bar{i}'.\, \phi_T^\star$ if and only if $\mathfrak{T}$ is $\neg\phi$-progressing. When the transition formula ϕ_T is specified with guarded commands, say with n commands $(A_1 \rhd B_1), \ldots, (A_n \rhd B_n)$, the safety property ϕ^* can be written as $\forall \bar{i}.\, \mathbf{G}\, (A_1^\star \vee \cdots \vee A_n^\star \vee \psi_2^\star)$. Notably, when both $\mathfrak{T}$ and $\neg\phi$ are abstractable (which is precisely the assumption of Theorem 6.3), the induced system $\mathfrak{T}^*$ and safety property ϕ^* are also abstractable. Consequently, the progress condition of $\mathfrak{T}$ with respect to $\neg\phi$ can be checked formally using the safety verification method stated in Theorem 6.2.

6.5 Liveness Verification under Fairness Requirements

We briefly discuss how to perform liveness verification of a subclass of array systems in the presence of fairness requirements. Such requirements are essential for specifying reactive and concurrent process systems (cf. [Demri et al. 2016; Manna and Pnueli 2012]).

Definition 6.4 (Index-bounded array system). An array system $(\mathcal{V}, \phi_I, \phi_T)$ is *index-bounded* if for each index variable $x \in \mathcal{V}$, there exist two parameters $l_x, h_x \in \mathcal{V}$ such that $s(l_x) \leq s(x) \leq s(h_x)$ holds in any reachable state s of the system. In other words, the valuation of x is bounded between l_x and h_x during system executions.

Intuitively, index-boundedness requires the range of index variables to be finite at runtime, while leaving the range of array elements unrestricted. This assumption is reasonable since in practice, index variables are mostly used to access elements of a finite array. Index-boundedness can be specified formally in the array system specification, or checked against a given array system by formulating the assumption as a safety array property.

Definition 6.5 (Fairness specification). An *(abstractable) fairness specification* is a temporal array property $\lambda := \forall i.\, \eta$ constrained by the grammar

$$\eta ::= \mathbf{GF}\,\psi \mid \mathbf{FG}\,\psi \mid \eta \vee \eta \mid \eta \wedge \eta,$$

where ψ is an abstractable indexed array formula. We say that a path π is *fair with respect to* λ, or simply *fair* when λ is clear, if π satisfies λ. A temporal array property ϕ holds for an array system under a fairness specification if the system does not have a fair path satisfying $\neg\phi$.

We note that our formalism of fairness is expressive enough to capture typical fairness requirements for reactive and concurrent systems, including *process fairness* (in form of $\mathbf{GF}\,\phi$), *weak fairness* (in form of $\mathbf{GF}\,\phi \vee \mathbf{GF}\,\psi$), and *strong fairness* (in form of $\mathbf{GF}\,\phi \vee \mathbf{FG}\,\psi$). See e.g. [Manna and Pnueli 2012] for a detailed discussion of these fairness requirements.

Given fairness specification λ and indexed predicates $\mathcal{P}$, we can compute a regular abstraction $(E, \mathcal{P})$ overapproximating λ in the abstract domain. Namely, if there is a fair concrete path $s_0\, s_1 \cdots$ with respect to λ, then there is a fair abstract path $t_0\, t_1 \cdots$ with respect to E, and $t_i \in \alpha(s_i)$ for each $i \geq 0$. When the abstract system is *weakly finite* [Esparza et al. 2012], that is, every infinite path eventually enters a cycle, checking the existence of a fair counterexample path essentially amounts to searching for a reachable fair cycle, which can in turn be reduced to a safety verification problem [Daniel et al. 2016; Schuppan and Biere 2006]. Indeed, for weakly finite systems, we can formally translate the abstract fairness condition $(E, \mathcal{P})$ to a formalism similar to the so-called *Büchi regular transition system (BRTS)* [Abdulla et al. 2012], which generalizes the translation from linear temporal properties to Büchi automata in finite-state model checking [Vardi and Wolper 1986]. We refer the interested reader to [Hong 2022] for the technical details of our translation procedure.

It is easy to see that regular abstractions of an index-bounded array system are weakly finite. Thus, together with Theorem 6.3, we can summarize our results in this section as follows.

Theorem 6.6. *Let $\mathfrak{T}$ be an index-bounded array system, ϕ be a liveness array property, and λ be a fairness specification. Suppose that $\mathfrak{T}$ and $\neg\phi$ are abstractable, and that $\mathfrak{T}$ is $\neg\phi$-progressing. Then we can compute finite automata $\mathcal{I}$, $\mathcal{T}$, and $\mathcal{F}$, as well as a regular fairness requirement Λ, such that ϕ holds for $\mathfrak{T}$ under the fairness specification Λ if the liveness property $(\mathcal{I}, \mathcal{T}, \mathcal{F})$ holds under the fairness requirement Λ.*

6.6 Verification of Monodic Temporal Properties

In this section, we briefly discuss a generic method to verify a monodic fragment of the temporal array properties. This method directly extends the classical tableaux-theoretic approach for LTL model checking (cf. [Clarke Jr et al. 2018]) to infinite structures. Similar techniques were employed in [Abdulla et al. 2012] for verifying LTL(MSO) properties of parameterized systems, and in [Padon et al. 2021] for verifying FO-LTL properties of first-order transition systems. The presentation here is essentially adopted from [Padon et al. 2021].

Consider a temporal array property ϕ over variables $\mathcal{V}$. We may eliminate $\mathbf{F}$ from $\neg\phi$ using the rule $\mathbf{F}\,\phi \equiv \neg\mathbf{G}\,\neg\phi$, and assume that $\mathbf{G}$ is the only temporal connective contained in $\neg\phi$. Let $sub(\neg\phi)$ denote the set of subformulae of $\neg\phi$. Define

$$\mathcal{V}_{\neg\phi} := \mathcal{V} \uplus \{\, g_\varphi : \mathbf{G}\,\varphi \in sub(\neg\phi) \,\},$$

where each g_φ is a fresh array variable. In a nutshell, to check that an array system satisfies ϕ, we take the product of the system and a "monitor" of $\neg\phi$ over the extended set of variables $\mathcal{V}_{\neg\phi}$. The product system will be associated with a fairness condition such that the projection of a *fair* path from $\mathcal{V}_{\neg\phi}$ to $\mathcal{V}$ coincides with a path satisfying $\neg\phi$ in the original array system. Thus, the

temporal property ϕ holds for the original system if and only if the product system has no fair path.

To apply the aforementioned reduction in our abstraction framework, however, the temporal property ϕ needs to meet some technical conditions. We say that ϕ is *monodic* [Hodkinson et al. 2000] if every *temporal* subformula of ϕ contains at most one free variable outside $\mathcal{V}$. For instance, when $\mathcal{V} = \{a, k\}$, the property $\forall j.\mathbf{G}\, a[j] > a[k]$ is monodic, whilst $\exists i.\forall j.\mathbf{G}\, a[j] > a[i]$ is not. We say that ϕ is *index-bounded* [Abdulla et al. 2016, 2012; Cimatti et al. 2022; Emerson and Namjoshi 2003] if there exists an index variable $h \in \mathcal{V}$ such that each quantified variable of ϕ is bounded above by h. This index-boundedness restriction may be imposed syntactically, for example, by replacing each subformula $\exists i.\psi$ of ϕ with $\exists i.\, i \le h \wedge \psi$, and each subformula $\forall i.\psi$ with $\forall i.\, i \le h \Rightarrow \psi$.

For a monodic formula ϕ over $\mathcal{V}$, we write $\phi(i)$ to indicate that i is the only free variable of ϕ outside $\mathcal{V}$, if there is any. In the case that ϕ has no such free variable, we just set i to be an arbitrary index variable outside $\mathcal{V}$. We use $\mathrm{FO}[\phi]$ to denote a first-order representation of ϕ, defined inductively as follows with $\delta \in \{\exists, \forall\}$:

$$\begin{aligned} \mathrm{FO}[\phi] &:= \phi \quad (\phi \text{ is non-temporal}) & \mathrm{FO}[\delta i.\, \phi] &:= \delta i.\, \mathrm{FO}[\phi] \\ \mathrm{FO}[\mathbf{G}\, \phi(i)] &:= g_\phi[i] \neq 0 & \mathrm{FO}[\phi_1 \vee \phi_2] &:= \mathrm{FO}[\phi_1] \vee \mathrm{FO}[\phi_2] \\ \mathrm{FO}[\neg\phi] &:= \neg\, \mathrm{FO}[\phi] & \mathrm{FO}[\phi_1 \wedge \phi_2] &:= \mathrm{FO}[\phi_1] \wedge \mathrm{FO}[\phi_2] \end{aligned}$$

Now, consider an index-bounded monodic property ϕ and an array system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$. Define a fair transition system $\mathfrak{T}_f := (\mathcal{V}_{\neg\phi}, \psi_I, \psi_T)$ with fairness requirement λ, where

$$\begin{aligned} \psi_I &:= \phi_I \wedge \mathrm{FO}[\neg\phi] \wedge \bigwedge\nolimits_{\mathbf{G}\varphi(i) \in sub(\neg\phi)} |g_\varphi| = h \\ \psi_T &:= \phi_T \wedge \forall i. \bigwedge\nolimits_{\mathbf{G}\varphi(i) \in sub(\neg\phi)} \Big(i > h \vee (g_\varphi[i] \neq 0 \Leftrightarrow (\mathrm{FO}[\varphi(i)] \wedge g'_\varphi[i] \neq 0))\Big) \\ \lambda &:= \forall i. \bigwedge\nolimits_{\mathbf{G}\varphi(i) \in sub(\neg\phi)} \mathbf{GF}\, (i > h \vee g_\varphi[i] \neq 0 \vee \neg\, \mathrm{FO}[\varphi(i)]) \end{aligned}$$

The fair transition system $\mathfrak{T}_f$ is constructed as the product of the original system $\mathfrak{T}$ and a monitor of $\neg\phi$ over the extended variables $\mathcal{V}_{\neg\phi}$. Given a subformula $\mathbf{G}\,\varphi$ of $\neg\phi$, the monitor updates the array g_φ along a path of $\mathfrak{T}$ in accordance with whether or not $\mathbf{G}\,\varphi$ is satisfied by the current path. More concretely, consider a path π of $\mathfrak{T}$ and some $t \in \mathbb{N}$. If $\pi \models \mathbf{G}\,\varphi(t)$, then the monitor maintains the array value $g_\varphi[t]$ along the path π to make sure $\varphi(t)$ is satisfied at every state of the path. Otherwise, if $\pi \not\models \mathbf{G}\,\varphi(t)$, then the monitor maintains the array value $g_\varphi[t]$ along the path π to make sure $\varphi(t)$ is falsified at some state of the path. In the second case, the fairness condition λ guarantees that the event of $\varphi(t)$ being falsified will not be postponed forever.

For a fair path of $\mathfrak{T}_f$, the projection of the path to the original variables $\mathcal{V}$ yields a path of the original system $\mathfrak{T}$ satisfying $\neg\phi$. Conversely, a path satisfying $\neg\phi$ in the original system $\mathfrak{T}$ can be lifted to the extended variables $\mathcal{V}_{\neg\phi}$ to obtain a fair path of $\mathfrak{T}_f$. As a consequence, one can verify the temporal array property ϕ for $\mathfrak{T}$ by checking fair termination of $\mathfrak{T}_f$.

Proposition 6.7 ([Abdulla et al. 2012; Padon et al. 2021]). *Let ϕ be an index-bounded monodic array property, $\mathfrak{T}$ be an array system, and $\mathfrak{T}_f$ be the fair transition system induced by $\mathfrak{T}$ and ϕ. Then ϕ holds for $\mathfrak{T}$ if and only if $\mathfrak{T}_f$ has no fair path.*

Furthermore, when $\mathfrak{T}$ is $\neg\phi$-progressing and the fair transition system $\mathfrak{T}_f$ is abstractable, termination analysis of $\mathfrak{T}_f$ is amenable to our liveness-to-safety reduction techniques in Section 6.5. This fact, in tandem with Theorem 6.6, leads to the following result:

Theorem 6.8. *Let ϕ be an index-bounded monodic array property, and $\mathfrak{T}$ be a $\neg\phi$-progressing index-bounded array system. If the fair transition system $\mathfrak{T}_f$ is abstractable, then we can compute finite automata $\mathcal{I}$ and $\mathcal{T}$ as well as a fairness requirement Λ, such that ϕ holds for $\mathfrak{T}$ if the transition system $(\mathcal{I}, \mathcal{T})$ always terminates under the fairness requirement Λ.*

7 CASE STUDIES

We present four case studies in this section: two array programs implementing selection sort and a simplified version of merge sort, and two distributed algorithms Dijkstra's self-stabilizing protocol and Chang-Roberts leader election protocol. To make our presentation succinct, we shall often omit the index range constraint in a formula when the range is clear from the context. For example, we shall write $\forall i.\, a[i] = 0$ instead of $\forall i.\, 1 \le i \wedge i \le |a| \Rightarrow a[i] = 0$.

7.1 Selection Sort and Merge Sort

For selection sort, we consider a selection sort program as follows:

$$pc = 0 \rhd low := 1,\ high := |a|,\ pc := 1$$
$$pc = 1 \wedge low \ge high \wedge 1 < high \rhd high := high - 1,\ low := 1$$
$$pc = 1 \wedge low < high \wedge a[low] \le a[high] \rhd low := low + 1$$
$$pc = 1 \wedge low < high \wedge a[low] > a[high] \rhd a[high] := a[low],\ a[low] := a[high],\ low := low + 1$$

The array transition system $\mathfrak{T} := (\mathcal{V}, \phi_I, \phi_T)$ induced by this program is

$$\mathcal{V} := \{pc, low, high, a, |a|\}, \quad \phi_I := pc = 0, \quad \phi_T := \phi_1 \vee \phi_2 \vee \phi_3 \vee \phi_4,$$

where $\phi_1, \phi_2, \phi_3, \phi_4$ are transition formulae derived from the four guarded commands. We verify the typical correctness properties of a sorting algorithm as follows.

Property	Specification	Explanation
P_1	$\mathbf{G}\,\forall i.\forall j.\, (i < j \wedge high < j \Rightarrow a[i] \le a[j])$	Sortedness
P_2	$(\forall i.\, a[i] = a_0[i]) \Rightarrow \mathbf{G}\,((\forall i.\exists j.\, a[i] = a_0[j]) \wedge (\forall i.\exists j.\, a_0[i] = a[j]))$	Permutation
P_3	$\mathbf{F}\, high \le 1$	Termination

Here, P_1 is a safety property stating that at any step, the array segment after the pivot position *high* is sorted; P_2 is a safety property stating that the array produced by the program has the same content as the input array modulo multiplicities; P_3 is a liveness property stating that the program eventually terminates. These three properties together establish the total correctness of the program for arrays with distinct values.

As for merge sort, we consider a nondeterministic merge sort program as follows:

$$pc = 0 \wedge sorted(low, mid) \wedge sorted(mid, high) \wedge \neg sorted(low, high) \wedge high \ge high_1 \rhd low_1 := low,\ pc := 1$$
$$pc = 1 \wedge low_1 < mid \wedge mid < high \wedge a[low_1] \le a[mid] \rhd low_1 := low_1 + 1$$
$$pc = 1 \wedge low_1 < mid \wedge mid < high \wedge a[low_1] > a[mid] \rhd a[n] = a[low_1],\ ptr := low_1,\ pc := 2$$
$$pc = 1 \wedge \neg(low_1 < mid \wedge mid < high) \rhd high_1 := high,\ low := *,\ mid := *,\ high := *,\ pc := 0$$
$$pc = 2 \wedge ptr < mid \rhd a[ptr + 1] := a[n],\ a[n] := a[ptr + 1],\ ptr := ptr + 1$$
$$pc = 2 \wedge ptr \ge mid \rhd a[low_1] := a[n],\ mid := mid + 1,\ low_1 := low_1 + 1,\ pc := 1$$

Here, $sorted(l, h) := \forall i.\, l \le i \wedge i < h - 1 \Rightarrow a[i] \le a[i + 1]$ is an auxiliary formula expressing that the array segment $a[l \ldots h)$ is sorted. This program starts with $pc = 0$ and $high_1 = 1$, and guesses the values of *low*, *mid*, and *high* within the range $\{1, \ldots, n\}$. If the program spots a merge

opportunity by this guess (i.e. the guard of the first command is satisfied), it proceeds to $pc = 1$ and performs an in-place array merge. After the merge, the program returns to $pc = 0$ and guesses the values of low, mid, and $high$ again. We verify the following eventuality property of the program:

$$P_1 := \mathbf{F}\,(pc = 0 \wedge (\neg sorted(low, mid) \vee \neg sorted(mid, high) \vee sorted(low, high) \vee high < high_1)),$$

which states that the program eventually fails to pinpoint a merge opportunity at $pc = 0$. When P_1 holds, any execution run that consistently spots a merge opportunity at $pc = 0$ eventually reaches $pc = 0$ with no further such opportunities, at which point the array segment $a[1 \dots n)$ is sorted. In other words, P_1 indicates that the program eventually produces a sorted array on the "proper" execution runs, namely, the runs where the program consistently attempts to merge partially sorted array segments.

7.2 Dijkstra's Self-Stabilizing Algorithm

Dijkstra's self-stabilizing algorithm [Dijkstra 1982], as introduced in Section 2, can be specified in our modeling language as follows:

$$pid = 1 \wedge a[pid] = a[n] \wedge a[pid] < k - 1 \rhd a[pid] := a[pid] + 1,\ pid := *$$
$$pid = 1 \wedge a[pid] = a[n] \wedge a[pid] = k - 1 \rhd a[pid] := 0,\ pid := *$$
$$pid > 1 \wedge pid \leq n \wedge a[pid] \neq a[pid - 1] \rhd a[pid] := a[pid - 1],\ pid := *$$

Here, we have identified $|a|$ with n and used pid to denote the ID of the process to be selected by the scheduler. For simplicity, we assume that only privileged processes can be scheduled. Dijkstra's algorithm is progressing under this scheduling assumption, since there always exists at least one privileged process in the system. (Recall that we have proved this fact in Example 5.1.)

Dijkstra's algorithm can be initialized with arbitrarily many privileged processes. When $n \leq k$, the system will converge to a stable state containing exactly one privileged process. For convenience, define an auxiliary formula $priv(i) := (i = 1 \wedge a[i] = a[n]) \vee (i \neq 1 \wedge a[i] \neq a[i - 1])$. We can then express the self-stabilizing property of Dijkstra's algorithm as

$$P := \mathbf{FG}\,(\exists i.\ priv(i) \wedge \forall j.\ j \neq i \Rightarrow \neg priv(j)),$$

meaning that the system eventually converges to exactly one privileged process and remains so forever. To prove the self-stabilizing property P, we create subgoals $P_1, \dots, P_4$ for the property as listed in the following table.

Property	Specification	Explanation
Q_1	$a[1] = a[n] \wedge pid = 1$	Process 1 is privileged and scheduled
Q_2	$\forall i.\ i \neq 1 \Rightarrow a[i] \neq a[1]$	$x_i \neq x_1$ for all $i \neq 1$
Q_3	$\forall i.\ i \neq 1 \Rightarrow a[i] = a[i - 1]$	Process i is unprivileged for all $i \neq 1$
Q_4	$\exists i.\ priv(i) \wedge \forall j.\ j \neq i \Rightarrow \neg priv(j)$	Exactly one process is privileged
P_1	$\mathbf{GF}\,Q_1$	Recurrence property
P_2	$\mathbf{GF}\,Q_1 \Rightarrow \mathbf{F}\,Q_2$	Liveness property under fairness condition
P_3	$Q_2 \Rightarrow \mathbf{F}\,(Q_1 \wedge Q_3)$	Liveness property
P_4	$(Q_1 \wedge Q_3) \Rightarrow \mathbf{G}\,Q_4$	Safety property
P	$\mathbf{FG}\,Q_4$	Self-stabilizing property

Intuitively, P_1 states that process p_1 is privileged and scheduled infinitely often; P_2 states that if process p_1 is privileged and scheduled infinitely often, then eventually variable x_1 differs from all the other variables; P_3 states that if x_1 differs from all the other variables at some point, then

eventually p_1 is the only privileged process; P_4 states that if p_1 is the only privileged process, then the system has stabilized. To verify that Dijkstra's algorithm is self-stabilizing, it suffices to check the properties $P_1, \ldots, P_4$ separately, since the self-stabilizing property P is subsumed by $P_1 \wedge P_2 \wedge P_3 \wedge P_4$.

7.3 Chang-Roberts Algorithm

The Chang-Roberts algorithm [Chang and Roberts 1979] is a ring-based leader election protocol. The algorithm assumes that each process has a unique ID, and that messages can be passed on the ring in the clockwise direction. At first, all processes are active. An active process becomes passive after it emits or forwards a message. The algorithm starts when an active process turns into an initiator and emits a message tagged with its ID to the next process. Let id_i denote the ID of Process i. When Process i receives a message tagged with id, it reacts in three cases:

- $id < id_i$: Process i will purge the message.
- $id > id_i$: Process i will forward the message and become passive.
- $id = id_i$: Process i will announce itself as a leader.

The rationale behind the protocol is that only the message tagged with the largest ID will complete the round trip and make its sender the leader. We specify the protocol as follows:

$$id[pid] < msg[pid+1] \wedge st[pid] = 0 \rhd st[pid] := 1,\ pid := *$$
$$id[pid] \geq msg[pid+1] \wedge st[pid] = 0 \rhd st[pid] := 1,\ msg[pid+1] := id[pid],\ pid := *$$
$$msg[pid] \geq id[pid] \wedge msg[pid] < msg[pid+1] \rhd msg[pid] := 0,\ st[pid] := 1,\ pid := *$$
$$msg[pid] \geq id[pid] \wedge msg[pid] \geq msg[pid+1] \rhd msg[pid] := 0,\ st[pid] := 1,\ msg[pid+1] := msg[pid],\ pid := *$$

These four commands correspond to four operations as follows: an initiator emits its ID, which is purged by the successor process; an initiator emits its ID, which is cached by the successor; a process forwards a message, which is purged by the successor; a process forwards a message, which is cached by the successor. As before, we use pid to denote the ID of the scheduled process. We use $st[i]$ to represent the status of Process i, with 0 and 1 standing for active and passive, respectively. We use $id[i]$ and $msg[i]$ to represent the ID and the message buffer, respectively, of Process i. We stipulate that all process IDs are positive, and $msg[i] = 0$ if and only if the message buffer of Process i is empty. When a process receives multiple messages, it keeps the one attached with the largest ID. Finally, we modify these commands to distinguish the case $pid = n$: for this case, we replace $pid + 1$ with 1. We consider two correctness properties of the Chang-Roberts algorithm:

Formula	Definition	Description
$largest(i)$	$\forall j.\ id[i] \geq id[j]$	Process i has the largest ID
$unique(i)$	$\forall j.\ j \neq i \Rightarrow id[i] \neq id[j]$	Process i has a unique ID
$elected(i)$	$msg[i] = id[i]$	Process i is elected as a leader

Property	Specification
P_1	$\forall i.\ \neg largest(i) \wedge unique(i) \Rightarrow \mathbf{G}\ \neg elected(i)$
P_2	$(\forall i.\ \mathbf{GF}\ pid = i) \Rightarrow (\forall i.\ largest(i) \wedge unique(i) \Rightarrow \mathbf{F}\ elected(i))$

Here, P_1 is a safety property saying that a process not holding the largest ID is never elected as a leader; P_2 is a liveness property saying that a process holding the largest ID is eventually elected

Table 2. Results of applying the L^* learning-based model checker [Chen et al. 2017] and the SLRP tool [Lin and Rümmer 2016] in the abstract analysis of the case studies. In the table, P_0 denotes the progress conditions, and $P_1, \ldots, P_4$ are the temporal properties defined in Section 7. The table also shows the indexed predicates we used to compute regular abstractions. Note that these predicates contain Skolem constants produced in the formulae rewriting step of our method (see Section 6). For each property, we present the total runtime (Total), the computation time of the initial and refined abstract systems (Model), and the time consumed by the model checker (Solver).

Safety Properties		L^*		
Name	Indexed Predicates	Total	Model	Solver
s.sort P_0	$a[\mathtt{i}] \le a[high]$, $a[\mathtt{i}] \le a[p]$	0.5s	0.0s	0.4s
s.sort P_1	$a[\mathtt{i}] \le a[high]$, $a[\mathtt{i}] \le a[p]$	3.7s	0.1s	3.4s
s.sort P_2	$a[\mathtt{i}] = a_0[p]$, $a_0[\mathtt{i}] = a[p]$	6.3s	0.3s	5.7s
m.sort P_0	$a[\mathtt{i}] \ge a[n]$, $a[\mathtt{i}] \le a[\mathtt{i}+1]$, $a[\mathtt{i}] \le a[mid]$	0.9s	0.1s	0.8s
Dijk. P_4	$a[\mathtt{i}] = a[\mathtt{i}-1]$, $a[\mathtt{i}] = a[n]$, $a[\mathtt{i}] = a[1]$	1.2s	0.1s	1.0s
C.-R. P_1	$id[\mathtt{i}] = id[p]$, $id[\mathtt{i}] < id[q]$, $msg[\mathtt{i}] = id[p]$, $msg[\mathtt{i}] < id[q]$	3.3s	0.4s	2.7s
C.-R. P_0	$msg[\mathtt{i}] \neq 0$, $st[\mathtt{i}] = 0$, $id[\mathtt{i}] < id[p]$, $msg[\mathtt{i}] < id[p]$, $msg[\mathtt{i}] = id[p]$	8.4s	0.9s	7.2s

Liveness Properties		L^*			SLRP		
Name	Indexed Predicates	Total	Model	Solver	Total	Model	Solver
s.sort P_3	no predicates	0.7s	0.4s	0.2s	1.5s	0.1s	1.3s
m.sort P_1	$a[\mathtt{i}] \ge a[n]$, $a[\mathtt{i}] \le a[\mathtt{i}+1]$, $a[\mathtt{i}] \le a[mid]$	5m47s	5m39s	5s	1.4s	0.1s	1.1s
Dijk. P_1	$a[\mathtt{i}] = a[\mathtt{i}-1]$, $a[\mathtt{i}] = a[n]$, $a[\mathtt{i}] = a[1]$	t.o.	–	–	12m54s	3s	12m50s
Dijk. P_3	$a[\mathtt{i}] = a[\mathtt{i}-1]$, $a[\mathtt{i}] = a[n]$, $a[\mathtt{i}] = a[1]$	t.o.	–	–	3m14s	2s	3m10s
Dijk. P_2	$a[\mathtt{i}] = z$, $a[\mathtt{i}] = a[1]$	9m13s	4s	9m06s	n/a	–	–
C.-R. P_2	$st[\mathtt{i}] = 0$, $id[\mathtt{i}] < id[p]$, $msg[\mathtt{i}] < id[p]$, $msg[\mathtt{i}] = id[p]$	3m04s	11s	2m51s	n/a	–	–

as a leader under the fairness requirement that each process is scheduled infinitely often. The validity of these properties relies on the assumption that the process IDs are unique. Thus, we need to specify this assumption in the properties.

8 OPTIMIZATION AND EVALUATION

$$\frac{a' = a\{I \leftarrow a[J]\} \quad a[J] \bowtie a[K] \quad I \neq K}{a'[I] \bowtie a'[K]} \qquad \frac{a' = a\{J \leftarrow e\} \quad a[I] \bowtie a[K] \quad J \neq I \quad J \neq K}{a'[I] \bowtie a'[K]}$$

$$\frac{x' = J \quad a[I] \bowtie a[K]}{a'[I] \bowtie a'[K]} \qquad \frac{a' = a\{I \leftarrow a[J]\} \quad a[J] \bowtie n}{a'[I] \bowtie n} \qquad \frac{a' = a\{J \leftarrow e\} \quad a[I] \bowtie n \quad I \neq J}{a'[I] \bowtie n}$$

$$\frac{x' = J \quad a[I] \bowtie b[K]}{a'[I] \bowtie b'[K]} \qquad \frac{a' = a\{I \leftarrow b[J]\} \quad b[J] \bowtie c[K]}{a'[I] \bowtie c'[K]} \qquad \frac{a' = a\{J \leftarrow e\} \quad a[I] \bowtie b[K] \quad I \neq J}{a'[I] \bowtie b'[K]}$$

Fig. 1. Example constraint templates for updating and copying array contents, where $\bowtie \in \{=, \neq, <, \le, >, \ge\}$. Each template consists of multiple *premises* and a *consequence*. The leftmost premise corresponds to an update (i.e. the assignment part of a guarded command), and the rest of the premises come from the guard. The consequence copies a predicate value from the current state to the next state. These templates exploit a simplifying assumption that one guarded command updates precisely one index variable or array element.

We have implemented a prototype to evaluate our approach over the case studies. The verification of an array system consists of two stages: (i) computing regular abstractions from the system specification, and (ii) performing abstract analysis based on these regular abstractions. When we compute a regular abstraction as described in Section 5.2, a crucial step is to compute the constraint $\mathrm{cstr}(\psi)$ at (8), which could be obtained by enumerating the minimal unsatisfiable cores (MUCs).

For this purpose, we first compute MUCs for ψ using the MUST tool [Bendík and Černá 2020]. Since the number of MUCs could be large, including all clauses induced by these MUCs in $\mathrm{cstr}(\psi)$ is generally impractical. Instead, our tool computes the constraint in an incremental manner: we sort these clauses by size, and include a clause only when the constraints generated by the smaller clauses lead to spurious counterexample or timeout in the abstract analysis. As our experimental results have shown, most of the properties in our case studies can be verified by including a moderate number of clauses in the state formulae abstractions.

For transition formulae abstractions, we additionally use templates to capture constraints involving array updates (Figure 1). We syntactically populate these templates with predicates and index terms in ψ and $\mathcal{P}$ to derive feasible clauses for $\mathrm{cstr}(\psi)$. To illustrate, suppose that $(a[\mathtt{i}] \leq a[n]) \in \mathcal{P}$ and $\psi := a[p] \leq a[n] \wedge a' = a\{i \leftarrow a[p]\}$. Then, the first template in Figure 1 syntactically derives a feasible clause $\{i = n,\ a'[i] \leq a'[n]\}$ for $\mathrm{cstr}(\psi)$. (To see this, note that the first template is in the form of $A \wedge B \wedge C \Rightarrow D$, which is equivalent to $A \wedge B \Rightarrow \{\neg C, D\}$. By instantiating A to $a' = a\{i \leftarrow a[p]\}$, B to $a[p] \leq a[n]$, C to $i \neq n$, and D to $a'[i] \leq a'[n]$, we obtain the desired clause.) Although such clauses may also be produced using the incremental method described earlier, we choose to generate them directly using templates as they are essential for computing precise transition formulae abstractions.

When computing an abstraction formula, we include singleton feasible clauses by default, and incrementally inject other nontrivial feasible clauses in the refinement loop. For transition formulae, we furthermore include clauses populated by templates. We convert the abstraction formulae of an array system specification into finite-state automata mechanically, leveraging Mona [Klarlund and Møller 2001] and Z3 [de Moura and Bjørner 2008]. These automata comprise an abstract regular transition system for the concrete specification. We then analyze this abstract system using suitable regular model checkers in the literature. For safety properties, we perform the abstract analysis using the safety verifier by [Chen et al. 2017], which employs the L^* learning algorithm to generate inductive invariants. For liveness properties, we apply two independent, fully automated techniques: one combines liveness-to-safety reduction techniques (see Section 6.5) with the safety verifier by Chen et al.; the other exploits the liveness verifier SLRP [Lin and Rümmer 2016], which essentially uses an SAT solver to synthesize well-founded relations. Note that our case studies contain two liveness properties with fairness requirements (Dijkstra P_2 and Chang-Roberts P_2). SLRP can only verify liveness under arbitrary schedulers and does not apply to these two properties.

We conducted the experiments on a laptop computer with a 3.6GHz Intel i7 processor, a 16GB memory limit, and a 20-minute timeout. Table 2 presents the results. For each of these properties, our prototype tool can compute sufficiently precise regular abstractions using constraints generated by no more than 10 clauses from the incremental and template-based methods. As for the abstract analysis, the safety properties turn out to be relatively easy: all of them can be proved by the L^* safety verifier in seconds. Regarding liveness properties, SLRP successfully proves all four liveness properties not requiring fair schedulers, while the L^* safety verifier proves only one of them, namely selection sort P_3, within the timeout. This property can be verified without using any indexed predicates; the resulting abstract system thus degenerates into an integer program. Reducing a liveness property to a safety property could be expensive: this operation accounts for the majority of the runtime for verifying merge sort P_1, and is likely the main bottleneck in the verification of Dijkstra P_1 and P_3 by L^*. Interestingly, the reduction takes a relatively short time for Dijkstra P_2 and Chang-Roberts P_2, and hence allows the safety verifier to prove them in time. Since these two properties hold only under fairness requirements, they cannot be handled by SLRP.

9 RELATED WORK

There is a huge body of research on the verification of array programs and systems. In this section, we provide some context for our work and discuss related work that has not yet been mentioned elsewhere in the paper.

Prior work exploiting predicate abstraction and interpolation exists in the context of verifying quantified inductive invariants for array programs [Alberti et al. 2012b; Cimatti et al. 2016; Jhala and McMillan 2007; Lahiri and Bryant 2007; McMillan 2008; Seghir et al. 2009]. *Indexed predicates*, which are essentially predicates containing free index variables, were first introduced by Flanagan and Qadeer [Flanagan and Qadeer 2002] to compose universally quantified inductive invariants. Lahiri and Bryant later extended and formalized this notion in the framework of *indexed predicate abstraction (IPA)* [Lahiri and Bryant 2004a,b, 2007]. IPA encodes abstract state sets as propositional formulae over Boolean variables, and universally quantifies index variables in the predicates when performing concretization. Since post-images of abstract states are generally not computable in this setting, Lahiri and Bryant devised overapproximations of them using a quantifier instantiation heuristic, which enabled the authors to reduce abstract reachability analysis to solving quantified Boolean formulae. In comparison, our abstraction framework encodes abstract state sets as first-order formulae over word variables, and exploits regular languages to overapproximate sets and relations in the abstract domain. These apparatuses allow us to reason about quantified formulae and temporal properties beyond those considered by Lahiri and Bryant. On the other hand, our abstraction function induces infinite abstract systems, for which we employ infinite-state model checking techniques (i.e. regular model checking [Abdulla 2012; Lin and Rümmer 2022]).

For a class of array systems that are used to model *multi-threaded programs*, specialized predicate abstraction techniques have been developed to infer universally quantified inter-thread properties. Many techniques along this line have focused on symmetric systems (i.e. the system behaves correctly regardless of thread arrangement [Basler et al. 2009; Donaldson et al. 2011, 2012; Pani et al. 2023]) and monotonic systems (i.e. the system is equipped with a well-quasi-ordering [Alberti et al. 2012a,b; Ranise and Ghilardi 2010]). Some techniques further abstract the target system into a symmetric/monotonic system by combining predication abstraction with some form of counter or monotonic abstraction (e.g. [Abdulla et al. 2010; Clarke et al. 2006, 2008; Ganjei et al. 2016; Kaiser et al. 2017]), thereby improving the expressiveness and effectiveness of the abstraction methods. Most of these techniques were designed for safety verification, and it is unclear whether they could be extended to handle liveness properties effectively.

Liveness verification of array systems is much more difficult than safety verification, and therefore has relatively fewer automatic techniques and tool supports. In the context of multi-threaded programs, *thread-modular analysis* [Cook et al. 2007; Ketema and Donaldson 2017; Malkis et al. 2007; Pani et al. 2021, 2023; Popeea and Rybalchenko 2012] is a popular verification methodology. The analysis considers each thread in isolation and overapproximates the behavior of the other threads by assuming that their effects are passive or irrelevant. Thread-modular analysis is therefore not suitable for verifying liveness properties that require coordination among threads. For example, in the case of Dijkstra's self-stabilizing algorithm, proving that Process 1 eventually updates its local variable requires showing that the other processes collectively make progress on updating their own variables. To reason about liveness properties relying on thread coordination, Farzan et al. [Farzan et al. 2016] introduced *well-founded proof spaces* and *quantified predicate automata (QPAs)* for parameterized verification. Like our approach, well-founded proof spaces take a language-theoretic view of termination and reduce checking termination to showing the absence

of lasso-shaped counterexample paths. The verification procedure provided by Farzan et al. explores symmetric proof spaces (i.e. proofs closed under permutation of thread identifiers). Also, the procedure requires to iteratively check language inclusion of QPAs, which is itself an undecidable problem and yet to have an effective implementation.

Ivy [McMillan and Padon 2018, 2020] is a deductive verification system that offers an automatic liveness-to-safety proof tactic for properties specified in LTL and a decidable fragment of pure first-order logic called *Effectively Propositional Logic (EPR)*. Specifically, Ivy reduces liveness to safety for infinite-state systems through *dynamic abstraction* [Padon et al. 2017, 2021], which is an overapproximation of cycle detection by dynamically choosing a finite projection of an infinite path. Despite the automatic reduction to safety, the resulting safety property still needs to be proved by the user. Moreover, Ivy does not support theory reasoning inherently, i.e., theories have to be encoded in EPR. Notably, in Ivy's abstraction scheme, the concrete specification, the abstract system, and the safety proof are all expressed in EPR. Our logical formalism is not limited to EPR and directly supports array logic with background element theories. (Our presentation has used the theory of Difference Arithmetic for array elements, as the theory already suffices to analyze our case studies. But in principle, it is easy to adapt our techniques to other background theories such as Linear Integer Arithmetic.)

10 CONCLUDING REMARKS

By combining indexed predicate abstraction, decision procedures, automatic structures, and regular model checking, we present a novel framework to verify linear-time properties of array systems. Given a first-order correctness specification, our framework provides a systematic method to compute regular overapproximations of the specification as a string rewriting system, which allows us to exploit a wealth of regular model checking techniques for analyzing both safety and liveness properties. Our experimental results show that this approach is able to verify non-trivial properties of array systems in several interesting case studies.

There are several immediate future research directions. Firstly, existing regular model checking techniques do not have good support for large alphabets. For this reason, the size of the resulting regular abstraction may grow exponentially in the number of predicates, especially with explicit representations of the letters (i.e. bitvectors). This leads to the following question: is it possible to extend regular model checking with symbolic representations of the alphabet symbols? We believe that the answer to this question is positive, given promising work in symbolic automata learning [Argyros and D'Antoni 2018; Drews and D'Antoni 2017] and bitvector theory SMT solving [Peled et al. 2023; Shi et al. 2021; Yao et al. 2020]. Secondly, our abstraction computation procedure currently uses simple incremental and template-based methods to search for constraints. We are confident that more sophisticated techniques such as the refinement-based search in IC3 [Cimatti et al. 2014; Komuravelli et al. 2015] can be integrated with our procedure for constraint discovery. Lastly, we have assumed in this work that appropriate indexed predicates have been supplied. Generating nontrivial indexed predicates for array properties is very challenging. In our case studies, we extract predicates from the atomic formulae of system and property specifications. Thus far, this suffices for our current goal, which has been to demonstrate the expressive power and viability of regular abstractions for array systems in interesting case studies. The next step of our research is, therefore, to study automatic generation and refinement of indexed predicates.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and corrections. Chih-Duo Hong is partly supported by the National Science and Technology Council, Taiwan, under grant

number 112-2222-E004-001-MY3. Anthony Lin is supported by European Research Council under European Union's Horizon 2020 research and innovation programme (grant agreement no 101089343).

REFERENCES

Martin Abadi. 1989. The power of temporal proofs. *Theoretical Computer Science* 65, 1 (1989), 35–83.

Parosh Aziz Abdulla. 2012. Regular Model Checking. *International Journal on Software Tools for Technology Transfer (STTT)* 14, 2 (2012), 109–118.

Parosh Aziz Abdulla, Yu-Fang Chen, Giorgio Delzanno, Frédéric Haziza, Chih-Duo Hong, and Ahmed Rezine. 2010. Constrained Monotonic Abstraction: A CEGAR for Parameterized Verification. In *International Conference on Concurrency Theory (CONCUR)*. Springer, 86–101.

Parosh Aziz Abdulla, Giorgio Delzanno, and Ahmed Rezine. 2009. Approximated Parameterized Verification of Infinite-State Processes with Global Conditions. *Formal Methods in System Design (FMSD)* 34, 2 (2009), 126–156.

Parosh Aziz Abdulla, Frédéric Haziza, and Lukáš Holík. 2016. Parameterized Verification Through View Abstraction. *International Journal on Software Tools for Technology Transfer (STTT)* 18, 5 (2016), 495–516.

Parosh Aziz Abdulla, Bengt Jonsson, Marcus Nilsson, Julien d'Orso, and Mayank Saksena. 2012. Regular Model Checking for LTL(MSO). *International Journal on Software Tools for Technology Transfer (STTT)* 14, 2 (2012), 223–241.

Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. 2012a. Lazy Abstraction with Interpolants for Arrays. In *International Conference on Logic Programming and Automated Reasoning (LPAR)*. Springer, 46–61.

Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. 2012b. SAFARI: SMT-Based Abstraction for Arrays with Interpolants. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 679–685.

Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. 2017. A Framework for the Verification of Parameterized Infinite-State Systems. *Fundamenta Informaticae* 150, 1 (2017), 1–24.

George Argyros and Loris D'Antoni. 2018. The Learnability of Symbolic Automata. In *International Conference on Computer Aided Verification (CAV)*. Springer, 427–445.

Gérard Basler, Michele Mazzucchi, Thomas Wahl, and Daniel Kroening. 2009. Symbolic counter abstraction for concurrent software. In *International Conference on Computer Aided Verification (CAV)*. Springer, 64–78.

Jaroslav Bendík and Ivana Černá. 2020. MUST: minimal unsatisfiable subsets enumeration tool. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 135–152.

Jaroslav Bendík, Ivana Černá, and Nikola Beneš. 2018. Recursive online enumeration of all minimal unsatisfiable subsets. In *Automated Technology for Verification and Analysis (ATVA)*. Springer, 143–159.

Michael Benedikt, Leonid Libkin, Thomas Schwentick, and Luc Segoufin. 2003. Definable Relations and First-Order Query Languages over Strings. *Journal of the ACM (JACM)* 50, 5 (2003), 694–751.

Roderick Bloem, Swen Jacobs, Ayrat Khalimov, Igor Konnov, Sasha Rubin, Helmut Veith, and Josef Widder. 2015. *Decidability of Parameterized Verification.* Morgan & Claypool Publishers.

Achim Blumensath and Erich Gradel. 2000. Automatic Structures. In *Symposium on Logic in Computer Science (LICS)*. IEEE, 51–62.

Achim Blumensath and Erich Grädel. 2004. Finite Presentations of Infinite Structures: Automata and Interpretations. *Theory of Computing Systems (TCS)* 37, 6 (2004), 641–674.

Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. 2004. Abstract Regular Model Checking. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 372–386.

Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. 2000. Regular Model Checking. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 403–418.

Aaron R. Bradley and Zohar Manna. 1998. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer.

Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. 2006. What's Decidable about Arrays?. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 427–442.

Ernest Chang and Rosemary Roberts. 1979. An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes. *Communications of the ACM (CACM)* 22, 5 (1979), 281–283.

Yu-Fang Chen, Chih-Duo Hong, Anthony W. Lin, and Philipp Rümmer. 2017. Learning to Prove Safety over Parameterised Concurrent Systems. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*. Springer, 76–83.

Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2014. IC3 Modulo Theories via Implicit Predicate Abstraction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Springer, 46–61.
Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. 2016. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods in System Design* 49 (2016), 190–218.
Alessandro Cimatti, Alberto Griggio, and Gianluca Redondi. 2022. Verification of SMT Systems with Quantifiers. In *International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Springer, 154–170.
Alessandro Cimatti, Alberto Griggio, Gianluca Redondi, et al. 2021. Universal Invariant Checking of Parametric Systems with Quantifier-free SMT Reasoning. In *International Conference on Automated Deduction (CADE)*. Springer, 131–147.
Edmund Clarke, Muralidhar Talupur, and Helmut Veith. 2006. Environment Abstraction for Parameterized Verification. In *International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 126–141.
Edmund Clarke, Murali Talupur, and Helmut Veith. 2008. Proving Ptolemy Right: The Environment Abstraction Framework for Model Checking Concurrent Systems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 33–47.
Edmund M. Clarke, Orna Grumberg, and Michael C. Browne. 1986. Reasoning about Networks with Many Identical Finite State Processes. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 240–248.
Edmund M. Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. 2018. *Model Checking*. MIT press.
Thomas Colcombet and Christof Löding. 2007. Transforming Structures by Set Interpretations. *Logical Methods in Computer Science (LMCS)* 3, 2 (2007), paper–4.
Byron Cook, Andreas Podelski, and Andrey Rybalchenko. 2007. Proving Thread Termination. In *Programming Language Design and Implementation (PLDI)*. ACM, 320–330.
Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 238–252.
Jakub Daniel, Alessandro Cimatti, Alberto Griggio, Stefano Tonetta, and Sergio Mover. 2016. Infinite-State Liveness-To-Safety via Implicit Abstraction and Well-Founded Relations. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 271–291.
Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 337–340.
Stéphane Demri, Valentin Goranko, and Martin Lange. 2016. *Temporal logics in computer science: finite-state systems*. Vol. 58. Cambridge University Press.
Edsger W. Dijkstra. 1982. Self-Stabilization in Spite of Distributed Control. In *Selected Writings on Computing: A Personal Perspective*. Springer, 41–46.
Alastair Donaldson, Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2011. Symmetry-aware predicate abstraction for shared-variable concurrent programs. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 356–371.
Alastair F. Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. 2012. Counterexample-Guided Abstraction Refinement for Symmetric Concurrent Programs. *Formal Methods in System Design (FMSD)* 41, 1 (2012), 25–44.
Samuel Drews and Loris D'Antoni. 2017. Learning Symbolic Automata. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Vol. 10205. Springer, 173–189.
E. Allen Emerson and Kedar S. Namjoshi. 2003. On Reasoning about Rings. *International Journal of Foundations of Computer Science* 14, 04 (2003), 527–549.
Javier Esparza, Andreas Gaiser, and Stefan Kiefer. 2012. Proving termination of probabilistic programs using patterns. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 123–138.
Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. 2016. Proving Liveness of Parameterized Programs. In *Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12.
Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. 2019. Quantified Invariants via Syntax-Guided Synthesis. In *Computer Aided Verification (CAV)*. Springer, 259–277.
Paolo Felli, Alessandro Gianola, and Marco Montali. 2021. SMT-based safety checking of parameterized multi-agent systems. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, Vol. 35. PKP Publishing, 6321–6330.
Tomáš Fiedor, Lukáš Holík, Petr Janků, Ondřej Lengál, and Tomáš Vojnar. 2017. Lazy Automata Techniques for WS1S. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 407–425.
Cormac Flanagan and Shaz Qadeer. 2002. Predicate Abstraction for Software Verification. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 191–202.
Zeinab Ganjei, Ahmed Rezine, Petru Eles, and Zebo Peng. 2016. Counting dynamically synchronizing processes. *International Journal on Software Tools for Technology Transfer (STTT)* 18 (2016), 517–534.

Yeting Ge and Leonardo De Moura. 2009. Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories. In *International Conference on Computer Aided Verification (CAV)*. Springer, 306–320.

Steven M. German and A. Prasad Sistla. 1992. Reasoning about Systems with Many Processes. *Journal of the ACM (JACM)* 39, 3 (1992), 675–735.

Silvio Ghilardi, Alessandro Gianola, and Deepak Kapur. 2021. Interpolation and Amalgamation for Arrays with MaxDiff. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. Springer, 268–288.

Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. 2016. SMT-Based Verification of Parameterized Systems. In *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 338–348.

Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. 2018. Quantifiers on Demand. In *Automated Technology for Verification and Analysis (ATVA)*. Springer, 248–266.

Peter Habermehl, Radu Iosif, and Tomáš Vojnar. 2008. A Logic of Singly Indexed Arrays. In *International Conference on Logic Programming and Automated Reasoning (LPAR)*. Springer, 558–573.

Ian Hodkinson, Frank Wolter, and Michael Zakharyaschev. 2000. Decidable Fragments of First-Order Temporal Logics. In *Annals of Pure and Applied Logic*. Elsevier, 181–185.

Jochen Hoenicke and Tanja Schindler. 2018. Efficient interpolation for the theory of arrays. In *International Joint Conference on Automated Reasoning (IJCAR)*. Springer, 549–565.

Chih-Duo Hong. 2022. *Symbolic techniques for parameterised verification.* Ph. D. Dissertation. University of Oxford.

Ranjit Jhala and Kenneth L. McMillan. 2007. Array Abstractions from Proofs. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 193–206.

Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. 2018. Predicate Abstraction for Program Verification: Safety and Termination. In *Handbook of Model Checking*. Springer, 447–491.

Alexander Kaiser, Daniel Kroening, and Thomas Wahl. 2017. Lost in abstraction: Monotonicity in multi-threaded programs. *Information and Computation* 252 (2017), 30–47.

Jeroen Ketema and Alastair F. Donaldson. 2017. Termination Analysis for GPU Kernels. *Science of Computer Programming* 148 (2017), 107–122.

Nils Klarlund and Anders Møller. 2001. *Mona Version 1.4: User Manual.* BRICS, Department of Computer Science, University of Aarhus Denmark.

Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. 2002. MONA Implementation Secrets. *International Journal of Foundations of Computer Science* 13, 04 (2002), 571–586.

Anvesh Komuravelli, Nikolaj S. Bjørner, Arie Gurfinkel, and Kenneth L. McMillan. 2015. Compositional Verification of Procedural Programs using Horn Clauses over Integers and Arrays. In *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 89–96.

Daniel Kroening and Ofer Strichman. 2016. *Decision Procedures.* Springer.

Shuvendu K. Lahiri and Randal E. Bryant. 2004a. Constructing Quantified Invariants via Predicate Abstraction. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Springer, 267–281.

Shuvendu K. Lahiri and Randal E. Bryant. 2004b. Indexed Predicate Discovery for Unbounded System Verification. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 135–147.

Shuvendu K. Lahiri and Randal E. Bryant. 2007. Predicate Abstraction with Indexed Predicates. *ACM Transactions on Computational Logic (TOCL)* 9, 1 (2007), 4–es.

Mark H. Liffiton, Alessandro Previti, Ammar Malik, and Joao Marques-Silva. 2016. Fast, flexible MUS enumeration. *Constraints* 21 (2016), 223–250.

Anthony W. Lin and Philipp Rümmer. 2016. Liveness of Randomised Parameterised Systems under Arbitrary Schedulers. In *International Conference on Computer-Aided Verification (CAV)*. Springer, 112–133.

Anthony W. Lin and Philipp Rümmer. 2022. Regular model checking revisited. In *Model Checking, Synthesis, and Learning: Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday*. Springer, 97–114.

Haojun Ma, Aman Goel, Jean-Baptiste Jeannin, Manos Kapritsos, Baris Kasikci, and Karem A Sakallah. 2019. I4: incremental inference of inductive invariants for verification of distributed protocols. In *The Symposium on Operating Systems Principles (SOSP)*. ACM, 370–384.

Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. 2007. Precise thread-modular verification. In *International Static Analysis Symposium (SAS)*. Springer, 218–232.

Makai Mann, Ahmed Irfan, Alberto Griggio, Oded Padon, and Clark Barrett. 2022. Counterexample-Guided Prophecy for Model Checking Modulo the Theory of Arrays. *Logical Methods in Computer Science (LMCS)* 18 (2022), 131–147.

Zohar Manna and Amir Pnueli. 2012. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Science & Business Media.

John McCarthy. 1993. Towards a mathematical science of computation. *Program Verification: Fundamental Issues in Computer Science* 1, 1 (1993), 35–56.

Kenneth L. McMillan. 2008. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 413–427.

Kenneth L. McMillan. 2018. Eager Abstraction for Symbolic Model Checking. In *International Conference on Computer Aided Verification (CAV)*. Springer, 191–208.

Kenneth L. McMillan and Oded Padon. 2018. Deductive Verification in Decidable Fragments with Ivy. In *International Static Analysis Symposium (SAS)*. Springer, 43–55.

Kenneth L. McMillan and Oded Padon. 2020. Ivy: A Multi-Modal Verification Tool for Distributed Algorithms. In *International Conference on Computer Aided Verification (CAV)*. Springer, 190–202.

Oded Padon, Jochen Hoenicke, Giuliano Losa, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2017. Reducing Liveness to Safety in First-Order Logic. *Symposium on Principles of Programming Languages (POPL)* 2, POPL (2017), 1–33.

Oded Padon, Jochen Hoenicke, Kenneth L. McMillan, Andreas Podelski, Mooly Sagiv, and Sharon Shoham. 2021. Temporal prophecy for proving temporal properties of infinite-state systems. *Formal Methods in System Design (FMSD)* 57 (2021), 246–269.

Thomas Pani, Georg Weissenbacher, and Florian Zuleger. 2021. Rely-guarantee bound analysis of parameterized concurrent shared-memory programs: With an application to proving that non-blocking algorithms are bounded lock-free. *Formal Methods in System Design (FMSD)* 57, 2 (2021), 270–302.

Thomas Pani, Georg Weissenbacher, and Florian Zuleger. 2023. Thread-modular counter abstraction: automated safety and termination proofs of parameterized software by reduction to sequential program verification. *Formal Methods in System Design (FMSD)* 60 (2023), 1–38.

Matan I Peled, Bat-Chen Rothenberg, and Shachar Itzhaky. 2023. SMT sampling via model-guided approximation. In *International Symposium on Formal Methods (FM)*. Springer, 74–91.

Corneliu Popeea and Andrey Rybalchenko. 2012. Compositional Termination Proofs for Multi-Threaded Programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 237–251.

Silvio Ranise and Silvio Ghilardi. 2010. Backward Reachability of Array-Based Systems by SMT Solving: Termination and Invariant Synthesis. *Logical Methods in Computer Science (LMCS)* 6, 4 (2010), 25–44.

Viktor Schuppan and Armin Biere. 2006. Liveness Checking as Safety Checking for Infinite State Spaces. *Electronic Notes in Theoretical Computer Science (ENTCS)* 149, 1 (2006), 79–96.

Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. 2009. Abstraction Refinement for Quantified Array Assertions. In *Static Analysis Symposium (SAS)*. Springer, 3–18.

Xiaomu Shi, Yu-Fu Fu, Jiaxiang Liu, Ming-Hsien Tsai, Bow-Yaw Wang, and Bo-Yin Yang. 2021. CoqQFBV: A Scalable Certified SMT Quantifier-Free Bit-Vector Solver. In *International Conference on Computer Aided Verification (CAV)*. Springer, 149–171.

Dirk van Dalen. 1994. *Logic and structure*. Vol. 3. Springer.

Moshe Y. Vardi and Pierre Wolper. 1986. An Automata-Theoretic Approach to Automatic Program Verification. In *Symposium on Logic in Computer Science (LICS)*. IEEE, 322–331.

Peisen Yao, Qingkai Shi, Heqing Huang, and Charles Zhang. 2020. Fast bit-vector satisfiability. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 38–50.

Received 2023-07-11; accepted 2023-11-07