J Autom Reasoning (2014) 52:379–405
DOI 10.1007/s10817-013-9293-6

# Array Theory of Bounded Elements and its Applications

**Min Zhou · Fei He · Bow-Yaw Wang · Ming Gu · Jiaguang Sun**

Received: 11 June 2012 / Accepted: 9 September 2013 / Published online: 24 September 2013
© Springer Science+Business Media Dordrecht 2013

**Abstract** We investigate a first-order array theory of bounded elements. This theory has rich expressive power that allows free use of quantifiers. By reducing to weak second-order logic with one successor (WS1S), we show that the proposed array theory is decidable. Then two natural extensions to the new theory are shown to be undecidable. A translation-based decision procedure for this theory is implemented, and is shown applicable to program verification.

**Keywords** Satisfiability modulo theories · Array theory · Program verification

This work was supported by the Chinese National 973 Plan under grant No. 2010CB328003, the NSF of China under grants No. 61272001, 60903030, 91218302, the Chinese National Key Technology R&D Program under grant No. SQ2012BAJY4052, the Tsinghua University Initiative Scientific Research Program, and the National Science Council of Taiwan projects No. NSC101-2221-E-001-006-, NSC102-2221-E-001-017-.

M. Zhou (✉) · F. He · M. Gu · J. Sun
School of Software, Tsinghua University, Beijing 100084, China
e-mail: zhoumin03@gmail.com

M. Zhou · F. He · M. Gu · J. Sun
Tsinghua National Laboratory for Information Science and Technology, Beijing, China

M. Zhou · F. He · M. Gu · J. Sun
Key Laboratory for Information System Security, MOE, Beijing, China

B.-Y. Wang
Academia Sinica, Taipei, Taiwan

## 1 Introduction

It is common in programming that we want to count the number of different elements in an array. Consider the following pseudo code:

```
Input a : array of [0..255]
for i ← 0 to 255 do b[i] ← 0;
for i ← 0 to |a| − 1 do b[a[i]] ← 1;
k ← 0;
for i ← 0 to 255 do
  k ← k + b[i];
```

where $a$ is an array of bytes and $|a|$ denotes the size of $a$. The array $b$ is a list of *indicators*. $b[v]$ indicates whether there is an element in $a$ with the value $v$. The time complexity of this program is $O(|a|)$.

After executing the program, we would like to see that the following property hold: $b[v] = 1$ if and only if there is an index $i$ such that $a[i] = v$. More formally, it must hold that

$$\forall v \in [0, 255].\ (b[v] = 1 \leftrightarrow (\exists i.a[i] = v))$$

This gives a formula in array theory.

Array theory in its most general form is undecidable. Various decidable fragments have been studied by researchers. Quantifier-free array theory fragments are considered in [25, 26], etc. In their works, arrays are often treated as uninterpreted functions. Existing decision procedures for uninterpreted functions can be used to decide the satisfiability of array formulas. In [3, 10], array formulas with restricted use of quantifiers are discussed. Instantiation strategies are proposed to convert quantified formulas to equisatisfiable quantifier-free ones. Array theories with restricted use of quantifiers could also be found in [14] and [2] where the satisfiability of array formulas is reduced to the reachability problem of counter automata. In these approaches, the decidable fragments are very restrictive. Arbitrary quantification induces undecidability, and nested reads is allowed in few of the fragments with quantifiers.

We consider the theory of array with bounded elements in this paper. In our theory, an array can have arbitrarily large but finite size. Its elements, however, must be bounded. Quantifiers can be freely used in our first-order theory. We show that the proposed array theory with bounded elements is decidable. Using this theory, program properties, which cannot be verified by existing fragments, can be specified and verified. Particularly, the sample formula is in our decidable fragment.

In our theory, bounded elements reflect data storage format in the physical world. In current computer systems, primitive data types are actually stored with a bounded size. For example, variables of the `int` type are actually stored in most systems as 32-bit integers. Our array theory of bounded elements is surely a realistic abstraction of the array data type.

Adopting bounded elements moreover relieves us from the imposed restriction of decidable fragments in previous theories. In the array theory of bounded elements, arbitrarily quantified first-order formulas are decidable. Nested reads do not induce undecidability either. Both can be freely used in array formulas without incurring computability issues.

In addition to its generality, another significant advantage of adopting the array theory of bounded elements is its simplicity. It is almost standard to establish the decidability result by reducing it to WS1S. Since decision procedures for WS1S are publicly available (for example, Mona in [20]), our reduction immediately gives a decision procedure for the array theory of bounded elements.

We also discuss two natural extensions to the array theory of bounded elements. Our conclusion is that both unbounded elements and linear arithmetic on indices would introduce undecidability. We show the arrays with unbounded elements is undecidable by reducing from the Hilbert's tenth problem. For arrays of bounded elements, the undecidability result of linear arithmetic on array indices is somewhat surprising. We show that multiplication is expressible with linear arithmetic and arrays of bounded elements. Hence the Hilbert's tenth problem can be reduced to the extended theory.

Towards the practical interests of the new theory, a translation-based decision procedure is implemented. We demonstrate its applications by verifying some array-manipulating programs, for instance, a sorting algorithm. Experimental results show this new theory is quite helpful in expressing and verifying complex array properties. Some of those properties can not be formalized or verified by existing array theories.

A preliminary version of this research was published in [29]. A main drawback of the previous work is its lack of necessary proofs for the translation as well as implementation for the proposed decision procedure. In this paper, we add the proofs that were ignored in [29]. Moreover, we report our new results on the implementation. With this implementation, we are able to show the usability of the proposed theory in program verification.

This paper is organized as follows: First, we give the preliminaries in Section 2. Then the array theory UABE is presented in Section 3. In Section 4, we show that the satisfiability of UABE is decidable by reducing it to WS1S. The translation procedure is also formalized. A couple of extensions are discussed in Section 5. It is shown that both extensions lead to undecidability. Experiments and applications in program verification are given in Section 6. Related works are compared in Section 7, followed by the conclusion.

## 2 Preliminaries

### 2.1 WS1S

Weak Second-order logic with One Successor (WS1S) was first discussed in [6]. It is a second order logic with the signature $\{=, \in, \mathcal{S}\}$. There are first-order and second-order variables in WS1S. Using $\mathcal{V}_1, \mathcal{V}_2$ to denote them respectively, the minimal syntax of WS1S is defined as:

$$\phi ::= (p = \mathcal{S}(q)) \mid p \in X \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\phi \mid \exists X.\phi, \qquad p, q \in \mathcal{V}_1, X \in \mathcal{V}_2$$

An interpretation is an assignment on variables. First-order variables are interpreted over the set of natural numbers $\mathbb{N}$ while second-order variables are interpreted over *finite* subsets of $\mathbb{N}$, i.e. $\mathcal{P}(\mathbb{N})$. $\mathcal{S}$ is the successor function on $\mathbb{N}$.

$\in$ is the membership relation on $\mathbb{N} \times \mathcal{P}(\mathbb{N})$. Given an interpretation $\sigma$, the semantics of WS1S is defined as:

$$\begin{array}{ll}
\sigma \models (p = \mathcal{S}(q)) & \Longleftrightarrow \sigma(p) = \sigma(q) + 1 \\
\sigma \models p \in X & \Longleftrightarrow \sigma(p) \in \sigma(X) \\
\sigma \models \neg\phi & \Longleftrightarrow \sigma \not\models \phi \\
\sigma \models \phi_1 \vee \phi_2 & \Longleftrightarrow \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models \exists p.\phi & \Longleftrightarrow \sigma[p \leftarrow i] \models \phi, \text{ for some } i \in \mathbb{N} \\
\sigma \models \exists X.\phi & \Longleftrightarrow \sigma[X \leftarrow M] \models \phi, \text{ for some } \textit{finite} \text{ set } M \subseteq \mathbb{N}
\end{array}$$

Based on the minimal syntax, other predicates such as "$<$", "$\subseteq$" can be defined:

$$\begin{array}{rl}
p < q & \Longleftrightarrow \forall X.(q \in X \wedge \forall r.(\mathcal{S}(r) \in X \rightarrow r \in X) \rightarrow p \in X) \\
X \subseteq Y & \Longleftrightarrow \forall p.(p \in X \rightarrow p \in Y)
\end{array}$$

A first-order variable plus a constant natural number is expressible by nesting "$\mathcal{S}$". I.e, $i + 3$ is equivalent to $\mathcal{S}(\mathcal{S}(\mathcal{S}(i)))$.

Büchi proved in [6] that WS1S is decidable. Technically, there is a correspondence between WS1S formulas and regular languages. Some verification tools are developed, such as Mona in [16, 20]. Given a WS1S formula, Mona is able to give a `satisfiable`, `valid` or `unsatisfiable` answer as well as a satisfying-example or a counter-example.[1]

## 2.2 Terminology

In this paper, the domains of arrays, array indices and array elements are denoted by $\mathbb{A}$, $\mathbb{N}$, and $\mathbb{Z}_n$ respectively. Array indices are natural numbers in $\mathbb{N} = \{0, 1, 2, \ldots\}$. Given a fixed $n$, array elements are fixed-size integers in $\mathbb{Z}_n = \{0, 1, \ldots, 2^n - 1\}$ and arrays are finite sequences of elements, i.e. $\mathbb{A} = \mathbb{Z}_n^*$. We assume for simplicity that $\mathbb{Z}_n$ contains only non-negative values. This is not a limitation because the element theory could be more general as long as it could be encoded in WS1S by bit blasting.

For $\lambda \in \{\mathbb{N}, \mathbb{Z}_n, \mathbb{A}\}$, we use $\mathcal{V}(\lambda)$ to denote the set of variables whose type is $\lambda$. Specially, $\mathcal{V} = \mathcal{V}(\mathbb{N}) \cup \mathcal{V}(\mathbb{Z}_n) \cup \mathcal{V}(\mathbb{A})$ is the set of all variables. For convenience, the following convention is followed throughout this paper:

$$\begin{array}{r}
i, j, k \in \mathcal{V}(\mathbb{N}) \\
x, y, z \in \mathcal{V}(\mathbb{Z}_n) \\
a, b \in \mathcal{V}(\mathbb{A}) \\
c \in \mathbb{N} \\
l \in \mathbb{Z}_n
\end{array}$$

Without causing ambiguity, Boolean values are expressed as 0 and 1.

Given an array variable $a$, we use $a[i]$ to denote the *array read* which returns the $i$-th element of $a$, and $a\{i \leftarrow x\}$ the *array write* which returns a new array obtained by replacing the $i$-th element with $x$.

[1] A satisfying-example is an interpretation that satisfies the formula while a counter-example is one that falsifies the formula. The result of `valid` means all interpretations are satisfying-examples while `unsatisfiable` means all interpretation are counter-examples. `satisfiable` means some interpretations satisfy the formula but some falsify it.

To distinguish functions and predicates on different domains, a subscript $\cdot_n$ is used to denote those on $\mathbb{Z}_n$. For instance, the addition function on $\mathbb{Z}_n$ is denoted by "$+_n$".

## 3 The Theory of UABE

The "*read-over-write*" axioms for array theory is introduced in [22] and accepted by almost all array theories:

$$\begin{aligned}\forall a, i, j, x. \quad & (i = j) \rightarrow a\{i \leftarrow x\}[j] = x \\ \wedge\ & (i \neq j) \rightarrow a\{i \leftarrow x\}[j] = a[j]\end{aligned}$$

UABE is a single dimensional extensional array theory. In our array theory, the index domain is not bounded while the element domain is bounded, so we call it Unbounded Array with Bounded Element (UABE). It consists of:

- the index theory $\mathcal{T}_{\mathbb{N}}$: whose domain is $\mathbb{N}$ and its signature is $\{\mathcal{S}, <, =\}$;
- the element theory $\mathcal{T}_{\mathbb{Z}_n}$: whose domain is $\mathbb{Z}_n$ and its signature is $\{+_n, <_n, =_n\}$;

### 3.1 Syntax

The minimal syntax of UABE is shown in Table 1. Predicates and functions on $\mathbb{Z}_n$ is denoted by the subscript $n$, for instance "$=_n$". A special predicate "$\simeq$" is defined on $\mathbb{N} \times \mathbb{Z}_n$, which establishes the equality between $\mathbb{N}$ and $\mathbb{Z}_n$.

The size of an array is considered as one of its attributes, denoted by $|a|$. Although in many other works, arrays are treated as uninterpreted functions for which the concept of *size* is not clear. We think array size is helpful for writing precise array properties. For instance, each array `a` has an attribute `a.length` in Java. The property that `a` is sorted in ascending order could be formulated as $\forall i, j : \mathbb{N}. ((0 \leq i < j < |a|) \rightarrow (a[i] \leq_n a[j]))$. Without this attribute, the sortedness property could only be discussed in an interval $[l, r]$ as in many related works.

In the index theory $\mathcal{T}_{\mathbb{N}}$, constant offsets are allowed, such as $i + 3$. But additions of two variables, such as $i + j$, are forbidden. We will show later that allowing addition of index variables leads to undecidability. UABE is *extensional*, i.e, equality over array variables is allowed. For instance, "$a$ and $b$ are identical arrays" could be formulated as $a = b$. The axiom of extensional is different from others because we take into consideration the size of arrays:

$$a = b \iff |a| = |b| \wedge \forall i.(i < |a| \rightarrow a[i] = b[i])$$

**Table 1** The minimal syntax of UABE

| Identifier | | Definition | Remarks |
|---|---|---|---|
| $\sim$ | $\in$ | $\{=, <\}$ | Comparisons on $\mathbb{N}$ |
| $\sim_n$ | $\in$ | $\{=_n, <_n\}$ | Comparisons on $\mathbb{Z}_n$ |
| $P$ | $:=$ | $x \sim_n y \mid x \sim_n l \mid x =_n y +_n z \mid x =_n a[i]$<br>$i \sim j \mid i \sim c \mid i = j + c \mid i = \lvert a \rvert$<br>$i \simeq x \mid a = b \mid a = b\{i \leftarrow x\}$ | Atomic formulas |
| $F$ | $:=$ | $P \mid \neg F \mid F_1 \wedge F_2 \mid \exists v : \lambda. F[v]$ | $v \in \mathcal{V}, \lambda \in \{\mathbb{N}, \mathbb{Z}_n, \mathbb{A}\}$ |

**Table 2** Semantics of UABE

| | | | |
|---|---|---|---|
| (1) | $V \models (x \sim_n y)$ | $\Longleftrightarrow$ | $\mu(x) \sim \mu(y)$ |
| (2) | $V \models (x \sim_n l)$ | $\Longleftrightarrow$ | $\mu(x) \sim l$ |
| (3) | $V \models (x =_n y +_n z)$ | $\Longleftrightarrow$ | $\mu(x) = \mu(y) + \mu(z)$ |
| (4) | $V \models (x =_n a[i])$ | $\Longleftrightarrow$ | $(\nu(i) < \lvert\tau(a)\rvert) \wedge (\mu(x) = \tau(a)_{\nu(i)})$ |
| (5) | $V \models (i \sim j)$ | $\Longleftrightarrow$ | $\nu(i) \sim \nu(j)$ |
| (6) | $V \models (i \sim c)$ | $\Longleftrightarrow$ | $\nu(i) \sim c$ |
| (7) | $V \models (i = j + c)$ | $\Longleftrightarrow$ | $\nu(i) = \nu(j) + c$ |
| (8) | $V \models (i = \lvert a\rvert)$ | $\Longleftrightarrow$ | $\nu(i) = \lvert\tau(a)\rvert$ |
| (9) | $V \models (i \simeq x)$ | $\Longleftrightarrow$ | $\nu(i) = \mu(x)$ |
| (10) | $V \models (a = b)$ | $\Longleftrightarrow$ | $\tau(a) = \tau(b)$ |
| (11) | $V \models (a = b\{i \leftarrow x\})$ | $\Longleftrightarrow$ | $\tau(a) = \tau(b)\{\nu(i) \leftarrow \mu(x)\}$ |
| (12) | $V \models \exists v.F[v]$ | $\Longleftrightarrow$ | $V \models F[v \leftarrow p]$, $p$ is type consistent with $v$ |
| (13) | $V \models \neg F$ | $\Longleftrightarrow$ | $V \not\models F$ |
| (14) | $V \models F_1 \wedge F_2$ | $\Longleftrightarrow$ | $(V \models F_1) \wedge (V \models F_2)$ |

Unlike most of other decidable array theories, our array theory allows arbitrary quantifiers on all index variables, element variables and array variables.

### 3.2 Semantics

A valuation[2] for UABE is a triple $V = (\mu, \nu, \tau)$, where

- $\mu : \mathcal{V}(\mathbb{Z}_n) \mapsto \mathbb{Z}_n$ assigns each element variable an integer in $\mathbb{Z}_n$,
- $\nu : \mathcal{V}(\mathbb{N}) \mapsto \mathbb{N}$ assigns each index variable a non-negative natural number,
- $\tau : \mathcal{V}(\mathbb{A}) \mapsto \mathbb{Z}_n^*$ assigns each array variable $a$ a *finite* sequence of $\mathbb{Z}_n$, i.e. $\tau(a) = a_0, a_1, \ldots, a_{\nu(\lvert a\rvert)-1}$.

Given a finite sequence $\tau(a)$, we use $\lvert\tau(a)\rvert$ to denote the length of it, $\tau(a)_p$ the $p$-th element of it, and $\tau(a)\{p \leftarrow v\}$ a new sequence by replacing the $p$-th element of $\tau(a)$ with $v$. Given a variable $v$ in UABE, the value of $v$ under the valuation $V$ is denoted by $V(v)$.

The semantics of UABE is listed in Table 2. (1) and (2) are comparisons between integer values. The semantics of adding two variables in $\mathbb{Z}_n$ is defined in (3). Note that $x, y, z$ are all $n$-bit integers, a necessary condition for $x =_n y +_n z$ evaluating to true is that the sum does not exceed the range of $n$-bit integers. The semantics enforces this because $x$ is also interpreted over $\mathbb{Z}_n$ and $\mu(x) < 2^n$. A note for (4) is that array accessed by indices that are equal to or larger than $\lvert a\rvert$ is not defined. For example, if $m \geq \lvert a\rvert$, then $x = a[m]$ does not hold for any $x$ since $a[m]$ is not defined. (8) actually defines the value of the variable $\lvert a\rvert$ to be the length of $\tau(a)$. With rule (9), a variable in $\mathbb{N}$ and a variable in $\mathbb{Z}_n$ can be compared. Hence, the index theory and the element theory are related, thus nested reads are possible. In rule (10), $a = b$ means the sequences of $\tau(a)$ and $\tau(b)$ are identical, which implies $\lvert\tau(a)\rvert = \lvert\tau(b)\rvert$. The semantics of existential quantifier is defined in (10). We say $p, v$ are type consistent, if $p \in \lambda$ and $v \in \mathcal{V}(\lambda)$ for any $\lambda \in \{\mathbb{N}, \mathbb{Z}_n, \mathbb{A}\}$.

[2]For UABE we say *valuation* to distinguish from *interpretation* of WS1S.

This semantic is defined conservatively. There are two kinds of undefined values in the semantics and both result in `false` when evaluating the truth value. Undefined values does not equal to any concrete value. Two undefined values are not equal, either.

- Array reads with out-of-bound index are undefined. In practice, it reflects memory access with an invalid address.
- The value of an arithmetic operation $x +_n y$ becomes undefined when $V(x) + V(y)$ exceeds $2^n$. In practice, it reflects possibly arithmetic overflow. For instance, it is not expected that $x +_n 1 =_n 0$ evaluates to `true` when $V(x) = 2^n - 1$. Usually, overflow implies design error in the program. Moreover, the element theory of UABE is actually parameterized. The addition $+_n$ could also be defined modulo $2^n$.

When an atomic formula evaluates to `false`, the reason is that either the valuation falsifies the formula or there is some term evaluates to undefined. This ambiguity is inconvenient in verification. However, undefined values are produced when some sanity conditions are violated. We can refine the formula by adding those conditions. For instance, $a[i] =_n x$ is refined to $i < |a| \wedge a[i] =_n x$ and the latter one could be used in verification.

Semantically, elements belonging to $a$ are indexed from 0 to $|a| - 1$. Two arrays are identical if they have the same size and their defined elements at the same position are equal.

The size of an array is interpreted as the length of the sequence. It is finite but could be arbitrarily large. For instance, the statement "for any natural number $m$, there is an array $a$ whose size is $m$ and all elements in $a$ are identically 0" can be expressed in UABE as:

$$\forall m : \mathbb{N}.\ (\exists a : \mathbb{A}.\ (|a| = m \wedge (\forall i : \mathbb{N}. i < |a| \rightarrow a[i] =_n 0)))$$

Therefore, each UABE may have infinitely many models and its decidability is not obvious.

### 3.3 Expressiveness

The syntax presented in Table 1 is minimal. Complex array properties can be formulated with the help of that syntax. This section briefly demonstrate the expressiveness of UABE:

- Boolean combinations of atomic formulas are expressible because $\{\neg, \wedge\}$ is sufficient to express $\{\vee, \rightarrow, \leftrightarrow\}$.
- Universal quantifier is expressible since $\forall v : \lambda. F[v] \iff \neg(\exists v : \lambda. \neg F[v])$.
- The minus function "$-$" is expressible since it can be converted to "$+$" by transposing the negative terms. e.g. $(x -_n y =_n z) \iff (x =_n y +_n z)$.
- Comparison operations $\{\leq, >, \geq\}$ on both index and element theory are expressible by using $\{\neg, <, +\}$.

Nested reads is expressible. Although index and element terms are of different types, we can compare them by using "$\simeq$". For example, the formula $z =_n a[a[i]]$ can be written as: $\exists v : \mathbb{N}.(z =_n a[v] \wedge v \simeq a[i])$.

For convenience, while writing a UABE formula, we feel free to use any syntax that has a logically equivalent formula in the minimal syntax. Then a lot of array properties can be formulated. Here we list some useful examples:

- Distinct: All elements in the array $a$ are distinct

$$\forall i, j : \mathbb{N}.(a[i] =_n a[j] \rightarrow i = j)$$

- Sorted: An array $a$ is sorted in ascending order

$$\forall i, j : \mathbb{N}.\ ((0 \leq i < j < |a|) \rightarrow (a[i] \leq_n a[j]))$$

- Monotonic: Elements in the array $a$ monotonically increase by 1 each step

$$\forall i : \mathbb{N}.(i + 1 < |a| \rightarrow a[i + 1] =_n a[i] +_n 1)$$

- Periodical: Elements in $a$ repeats periodically with period $T$

$$\forall i : \mathbb{N}.\ (i + T < |a| \rightarrow a[i] =_n a[i + T])$$

- Partitioned: Elements with indices less than $p$ are no larger than those with indices greater or equal to $p$

$$\forall i, j : \mathbb{N}.\ (i < p \leq j < |a| \rightarrow a[i] \leq_n a[j])$$

- Fibonacci array: The array $a$ is a Fibonacci array

$$(a[0] =_n 1) \wedge (a[1] =_n 1) \wedge \forall i : \mathbb{N}.\ (i + 2 < |a| \rightarrow a[i + 2] =_n a[i] +_n a[i + 1])$$

- Existence of the maximum element

$$|a| \geq 0 \rightarrow \exists x.\ ((\exists i.a[i] =_n x) \wedge \forall 0 \leq i < |a|.a[i] \leq x)$$

- Strictly sorted arrays are distinct

$$\forall i, j : \mathbb{N}.\ ((0 \leq i < j < |a|) \rightarrow (a[i] <_n a[j])) \rightarrow \forall i, j : \mathbb{N}.(a[i] =_n a[j] \rightarrow i = j)$$

- Also the example at the beginning of this paper

$$\forall v : \mathbb{Z}_n.\ \big((0 \leq v \leq 255) \rightarrow \big((\exists v' : \mathbb{N}.v' \simeq v \wedge b[v'] =_n 1) \leftrightarrow (\exists i : \mathbb{N}.a[i] =_n v)\big)\big)$$

There is no limit or restriction on using of quantifiers in UABE. We will see in Section 6 that it is rather useful in program verification because many verification conditions can be expressed directly.

## 4 Decidability

In this section, we show that the satisfiability and validity problem of UABE are decidable and present a decision procedure by translating UABE formulas into WS1S. For simplicity, it suffices to translate the minimal syntax.

### 4.1 Encoding of $\mathbb{N}$, $\mathbb{Z}_n$ and $\mathbb{A}$ Types in WS1S

Constants and variables of $\mathbb{N}$ type can be encoded directly because natural numbers are in the domain of WS1S. This section mainly focuses on encoding $\mathbb{Z}_n$ and $\mathbb{A}$ types.

Notice that $\mathbb{Z}_n$ elements can be viewed as $n$-bit non-negative integers. Consider a constant or a variable $t$ of type $\mathbb{Z}_n$. In the binary view, $t = \overline{t^{[n-1]}t^{[n-2]}\cdots t^{[0]}}$, where $t^{[d]}$ is the $d$-th bit for $d = 0, 1, \ldots, n-1$ and $t^{[n-1]}$ is the most significant bit. In WS1S, $t$ is encoded as a single set $S_t$ such that

$$S_t = \{d \mid t^{[d]} = 1\}$$

For instance, assume $n = 4$, the value $x = 5$ (in binary, $\overline{0101}$) is encoded as $S_x = \{0, 2\}$.

For $\mathbb{A}$ type variables and constants, they are viewed as sequences of $\mathbb{Z}_n$ elements. Their encoding is different. Given an array $a$, it is encoded as $n$ finite sets $a^{(0)}, a^{(1)}, \ldots, a^{(n-1)}$ and one natural number $|a|$. Each set $a^{(d)}$ consists of all indices of array elements whose $d$-th bit is 1. More formally:

$$a^{(d)} = \{i \mid a[i]^{[d]} = 1\}$$

Notice that we are encoding each *bit* of all elements as a set. For example, assume $n = 4$, an array $a$ with the size of 2 and $a[0] = 1$, $a[1] = 5$ is encoded as $a^{(0)} = \{0, 1\}$, $a^{(1)} = \emptyset$, $a^{(2)} = \{1\}$ and $a^{(3)} = \emptyset$.

**Definition 1** (Value Encoding Function: $\delta$) Given a UABE formula, the free variable set is $Y$ and they are encoded in WS1S as $Y'$. A value encoding function is a function $\delta$ that translate a valuation $V$ (on $Y$) to an interpretation $\sigma_V$ (on $Y'$) such that:

- For each variable $i \in Y \cap \mathcal{V}(\mathbb{N})$: assume $i$ is encoded by $i'$. Then it must hold that $\sigma_V(i') = V(i)$.
- For each variable $x \in Y \cap \mathcal{V}(\mathbb{Z}_n)$, assume $x$ is encoded by $S_x$. Then it must hold that $\sigma_V(S_x) \subseteq \{0, 1, \ldots, n-1\}$ and $\forall 0 \le d < n.(d \in \sigma_V(S_x) \leftrightarrow (V(x)^{[d]} = 1))$.
- For each variable $a \in Y \cap \mathcal{V}(\mathbb{A})$: assume $a$ is encoded by $|a|, a^{(0)}, \ldots, a^{(d-1)}$. Then it must hold that $\sigma_V(|a|) = V(|a|)$, for $0 \le d < n$, $\sigma_V(a^{(d)}) \subseteq \{0, 1, 2, \ldots, V(|a|) - 1\}$ and

$$\forall 0 \le d < n, 0 \le i < \sigma_V(|a|).(i \in \sigma_V(a^{(d)})) \leftrightarrow (V(a[i])^{[d]} = 1)$$

Here $\sigma_V = \delta(V)$. Intuitively, $\delta$ is a function that translate a valuation of UABE to an interpretation of WS1S according to the encoding.

### 4.2 Encoding of Predicates and Functions in WS1S

Predicates and functions on $\mathbb{N}$ are encoded directly in WS1S. Moreover, the only predicate over arrays is equality and it can be reduced to the equality over $\mathbb{Z}_n$ elements. So, in this section, we focus on predicates and functions in $\mathbb{Z}_n$.

Predicates on $\mathbb{Z}_n$ are $\{=_n, <_n\}$ and the only function is $\{+_n\}$. We assume that $x$, $y$, $z$ are constants or variables of $\mathbb{Z}_n$ type and $x^{[d]}$ is the $d$-th bit of $x$ (similarly for $y$ and $z$). Based on the discussion in the previous subsection, $x^{[d]}$ can be encoded as a

well-formed WS1S expression. For the predicates "$=_n$" and "$<_n$", they can be encoded in WS1S by bit-wise comparisons.

– Equality relation $x =_n y$ can be encoded in WS1S:

$$P_=(x, y) \equiv \bigwedge_{d=0}^{n-1} \left(x^{[d]} \leftrightarrow y^{[d]}\right)$$

– Order relation $x <_n y$ can be encoded in WS1S:

$$P_<(x, y) \equiv \bigvee_{d=0}^{n-1} \left( (\neg x^{[d]} \wedge y^{[d]}) \wedge \left( \bigwedge_{d'=d+1}^{n-1} (x^{[d']} \leftrightarrow y^{[d']}) \right) \right)$$

Addition function "$+_n$" is only allowed in the form of: $x =_n y +_n z$. Instead of defining a function, we define a special predicate which formulates the ripple-carry adder:

$$\begin{aligned} P_+(x, y, z) \equiv \exists C.( & \textstyle\bigwedge \neg(0 \in C) \wedge \neg(n \in C) \\ & \textstyle\bigwedge_{d=0}^{n-1}(d+1 \in C \leftrightarrow d \in C \wedge y^{[d]} \vee d \in C \wedge z^{[d]} \vee y^{[d]} \wedge z^{[d]}) \\ & \textstyle\bigwedge_{d=0}^{n-1}(x^{[d]} \leftrightarrow \neg(d \in C \leftrightarrow \neg(y^{[d]} \leftrightarrow z^{[d]})))) \end{aligned}$$

These predicates are defined intuitively. Using $V \models_{\mathsf{UABE}} f$ to denote that an UABE formula $f$ evaluates to *true* under the valuation $V$ (similarly for "$\models_{\mathsf{WS1S}}$"), the following lemmas hold.

**Lemma 1** *For all $x$, $y$ of $\mathbb{Z}_n$ type, all valuation $V$, it must hold*

$$\begin{aligned} V \models_{\mathsf{UABE}} (x =_n y) &\iff \delta(V) \models_{\mathsf{WS1S}} P_=(x, y) \\ V \models_{\mathsf{UABE}} (x <_n y) &\iff \delta(V) \models_{\mathsf{WS1S}} P_<(x, y) \end{aligned}$$

*Proof* $P_=$ and $P_<$ are just encoding of bit-wise comparisons, the conclusion is obvious. □

**Lemma 2** *For all $x$, $y$ and $z$ of $\mathbb{Z}_n$ type, all valuation $V$:*

$$V \models_{\mathsf{UABE}} (x =_n y +_n z) \iff \delta(V) \models_{\mathsf{WS1S}} P_+(x, y, z)$$

*Proof* $P_+(x, y, z)$ formulates a ripple-carry adder. In its definition, $C$ is the add-carry. The first line restricts the sum not to exceed $2^n - 1$. The second line constrains $C^{[d+1]} = 1$ if and only if at least two of $y^{[d]}$, $z^{[d]}$, $C^{[d]}$ are 1. The third line constrains $x^{[d]} = (y^{[d]}$ xor $z^{[d]}$ xor $C^{[d]})$. It is the encoding of binary (equality with) addition. □

In WS1S, user can not define new predicates. However, $P_=(x, y)$, $P_<(x, y)$ and $P_+(x, y, z)$ are just defined terms. They can be replaced by their body of definition. The resulted formulas are also well-formed in WS1S.

### 4.3 Translation

A translation rule is denoted by a horizontal line which separates the original and translated formula. Above the line is the original formula in UABE, while the translated formula in WS1S is shown below the line.

In the translation, each index variable $i$ in UABE is translated to a variable in WS1S (also denoted by $i$); each element variable $x$ is translated to a set $S_x$ and each array variable $a$ corresponds to a variable $|a|$ and $n$ sets: $a^{(0)}, \ldots, a^{(n-1)}$.

Our translation is applied in 2 phases: (1) translating the atomic formulas; (2) translating the quantifiers and Boolean structures.

### *4.3.1 Translation of Atomic Formulas*

Translation rules for atomic formulas are listed in Table 3. Above the lines are all the possible atomic formulas in the minimal syntax. Each rule is tagged with a number. Rule $(x.y)$ in this table corresponds to item $(x)$ in Table 2. Among those rules, (1.1), (1.2), (2.1), (2.2), (3), (4) are comparisons or addition on $\mathbb{Z}_n$, defined with the help of predefined terms $P_=$, $P_<$ and $P_+$. In (4), we add the sanity condition for array reads, $i < |a|$. (5), (6), (7), (8) are translated without modification since they are already well-formed in WS1S. For (9), to translate the atomic formula $i \simeq x$, the difficulty is type inconsistency between $i$ and $x$: $i \in \mathcal{V}(\mathbb{N})$ but $x \in \mathcal{V}(\mathbb{Z}_n)$. With the help of a specialized array $A_\simeq$, we replace the comparison between $i$ and $x$ to that of $A_\simeq[i]$ and $x$. In (10), $a = b$ implies that the size of $a, b$ are identical. In (11), array $a$ and $b$ are identical except at index $i$.

The array $A_\simeq$ is used to establish the connection between $\mathbb{N}$ and $\mathbb{Z}_n$ types. Notice that $\mathbb{Z}_n \subseteq \mathbb{N}$. $A_\simeq$ is defined such that for indices $i < 2^n$, $i$ and $A_\simeq[i]$ are equal in value. To encode $A_\simeq$ in WS1S, fortunately, we can avoid defining its elements one by one. By observation, $A_\simeq^{(d)}$ is a set which does not contain $\{0, \ldots, 2^d - 1\}$ but contains $\{2^d, \ldots, 2^{d+1} - 1\}$. Furthermore, it is periodical: for numbers $i \geq 2^{d+1}$, $i \in A_\simeq^{(d)} \iff (i - 2^{d+1}) \in A_\simeq^{(d)}$. For example, $A_\simeq^{(0)}$ contains all the 0-th (lowest) bit of array $[0, 1, 2, \ldots, 2^n - 1]$, i.e. $A_\simeq^{(0)} = \{1, 3, 5, 7, \ldots\}$. Thus $A^{(d)}$ is defined directly in WS1S:

$$\begin{aligned}
&\forall i.\, \big((i < 2^d \to i \notin A_\simeq^{(d)}) \wedge (2^d \leq i \wedge i < 2^{d+1} \to i \in A_\simeq^{(d)})\big) \\
\bigwedge\ &\forall i.\, \big(i + 2^{(d+1)} < 2^n \to ((i \in A_\simeq^{(d)}) \leftrightarrow (i + 2^{(d+1)} \in A_\simeq^{(d)}))\big) \\
\bigwedge\ &\forall i.\, \big(i \geq 2^n \to i \notin A_\simeq^{(d)}\big)
\end{aligned}$$

Use $\Theta$ to denote the conjunction of $|A_\simeq| = 2^n$ and all the definition of $A^{(d)}$ $(d = 0, 1, \ldots, n-1)$.

**Lemma 3** *Formula $\Theta$ is satisfiable.*

**Table 3** Translation rules for atomic formulas

| | | | | |
|---|---|---|---|---|
| (1.1) $\dfrac{x =_n y}{P_=(x, y)}$ | (1.2) $\dfrac{x <_n y}{P_<(x, y)}$ | (2.1) $\dfrac{x =_n l}{P_=(x, l)}$ | (2.2) $\dfrac{x <_n l}{P_<(x, l)}$ | (3) $\dfrac{x =_n y +_n z}{P_+(x, y, z)}$ |
| (4) $\dfrac{x =_n a[i]}{(i < \lvert a\rvert) \wedge P_=(x, a[i])}$ | (5) $\dfrac{i \sim j}{i \sim j}$ | (6) $\dfrac{i \sim c}{i \sim c}$ | (7) $\dfrac{i = j + c}{i = j + c}$ | (8) $\dfrac{i = \lvert a\rvert}{i = \lvert a\rvert}$ |
| (9) $\dfrac{i \simeq x}{i < 2^n \wedge P_=(A_\simeq[i], x)}$ | (10) $\dfrac{a = b}{(\lvert a\rvert = \lvert b\rvert) \wedge (\forall i.(i < \lvert a\rvert) \to P_=(a[i], b[i]))}$ | | | |
| (11) $\dfrac{a = b\{i \leftarrow x\}}{(i < \lvert a\rvert) \wedge P_=(a[i], x) \wedge \lvert a\rvert = \lvert b\rvert \wedge (\forall j.(j \neq i \wedge j < \lvert a\rvert) \to P_=(a[j], b[j]))}$ | | | | |

*Proof* Let $V(A_{\simeq}) = [0, 1, \ldots, 2^n - 1]$, Then $\delta(V) \models_{\mathsf{WS1S}} \Theta$. □

Denote the interpretation above that satisfies $\Theta$ by $\sigma_A$, i.e. $\sigma_A \models_{\mathsf{WS1S}} \Theta$. Denote the translation procedure for atomic formulas by $\Delta_a(\cdot)$. Given two interpretations $\alpha$ and $\beta$, use $\alpha \circ \beta$ to denote the composed interpretation such that for all $x$, $(\alpha \circ \beta)(x) = \alpha(\beta(x))$. Notice that if $\alpha$ and $\beta$ are defined on disjoint variable sets, then $\alpha \circ \beta = \beta \circ \alpha$.

**Lemma 4** *For all valuations V,* $\sigma_A \circ \delta(V) \models_{WS1S} \Theta$.

*Proof* Because $\Theta$ is defined on $|A_{\simeq}|$ and $A_{\simeq}^{(d)}$ only. For each free variable $v$ in $\Theta$, $(\sigma_A \circ \delta(V))(v) = \sigma_A(v)$. We already know $\sigma_A \models_{\mathsf{WS1S}} \Theta$, therefore this lemma holds. □

**Theorem 1** *If P is an atomic formula, then for all valuation V:*

$$V \models_{\mathsf{UABE}} P \iff \sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} (\Theta \wedge \Delta_a(P))$$

*Proof* By Lemmas 1 and 2, we know that $V \models_{\mathsf{UABE}} P \iff \delta(V) \models_{\mathsf{WS1S}} \Delta_a(P)$ holds for rules (1.1), (1.2), (2.1), (2.2), (3). It also holds for rules (4), (5), (6), (7) and (8) according to the semantics. Furthermore, $V \models_{\mathsf{UABE}} P \iff \sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Delta_a(P)$ for all these rules.

For the rule (9), by construction, we know for all $0 \leq l < 2^n$, $\sigma_A \models_{\mathsf{WS1S}} \Delta_a(l \simeq A_{\simeq}[l])$. If $V$ is a model for $i \simeq x$, then $i < 2^n$ should hold and $A_{\simeq}[i]$, $x$ should be equal, which means $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} i < 2^n$ and $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} P_n(A_{\simeq}[i], x)$. Thus $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Delta_a(i \simeq x)$. Along with the fact $\sigma_A \models_{\mathsf{WS1S}} \Theta$, the theorem holds.

According to the semantics of equalities over array variables and array writes, it can be proven that the theorem holds for rules (10) and (11). □

### *4.3.2 Translation of Complex Formulas*

Translation of quantifiers and Boolean structures are described recursively. Use $\Delta(\cdot)$ to denote the total translation function. If $F$ is an atomic formula, then $\Delta(F) = \Delta_a(F)$, otherwise $\Delta(F)$ is obtained by exhaustively applying the translation rules to $F$. Among the rules listed in Table 4, (12.1), (12.2) and (12.3) handle quantifiers and (13), (14) handle Boolean structures of formulas.

**Theorem 2** *Given any UABE formula f and any valuation V,*

$$V \models_{UABE} f \iff \sigma_A \circ \delta(V) \models_{WS1S} \Theta \wedge \Delta(f)$$

**Table 4** Translation rule for complex formulas

| | | |
|---|---|---|
| (12.1) $\dfrac{\exists x.F}{\exists S_x.\Delta(F)}$ | (12.2) $\dfrac{\exists i.F}{\exists i.\Delta(F)}$ | (12.3) $\dfrac{\exists a.F}{\exists \left(\lvert a\rvert, a^{(0)}, \ldots, a^{(n-1)}\right).\Delta(F)}$ |
| (13) $\dfrac{\neg F}{\neg\Delta(F)}$ | (14) $\dfrac{F_1 \wedge F_2}{\Delta(F_1) \wedge \Delta(F_2)}$ | |

*Proof* It suffices to show that $V \models_{\mathsf{UABE}} f$ if and only if (i) $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Theta$ and (ii) $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Delta(f)$. Lemma 4 ensures (i), then we prove (ii) by induction.

- If $f$ is an atomic formula, (ii) holds by Theorem 1.
- If $f$ is an existential quantified formula $\exists x.g$. It is translated to $\exists S_x.\Delta(g)$. The inductive hypothesis is $V \models_{\mathsf{UABE}} g \iff \sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Delta(g)$. It is known that $x$ is a free variable in $g \iff S_x$ is a free variable in $\Delta(g)$. Then $V \models_{\mathsf{UABE}} \exists x.g \iff$ there exists a constant $c$ such that $V \models_{\mathsf{UABE}} g[x/c] \iff$ there exists a constant $S_c$ such that $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Delta(g)[S_x/S_c] \iff \sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \exists S_x.\Delta(g)$. Here $g[x/c]$ is the formula obtained by replacing all free occurrences of $x$ to $c$ in $g$. $S_c$ is obtained by encoding $c$.
- If $f$ is a negation $\neg g$. It is translated to $\Delta(\neg g) = \neg\Delta(g)$. From the inductive hypothesis we know $V \models_{\mathsf{UABE}} g \iff \sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Theta \wedge \Delta(g)$. Along with Lemma 4, we know $V \models_{\mathsf{UABE}} \neg g \iff \sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Theta \wedge \Delta(\neg g)$.
- If $f$ is a conjunction $g_1 \wedge g_2$, the conclusion can be proven similarly.

The Boolean structure of any formula has finite levels, therefore we know that (ii) holds for all well-formed formulas. □

Theorem 2 explains the fact that given a UABE formula $f$ and a valuation $V$, we can always find an interpretation $\sigma_V$ for $\Theta \wedge \Delta(f)$ such that $V \models_{\mathsf{UABE}} f$ iff $\sigma_V \models_{\mathsf{WS1S}} \Theta \wedge \Delta(f)$.

Similarly, we can define the value decoding function which maps an interpretation of WS1S to a valuation of UABE.

**Definition 2** (Value Decoding Function: $\delta^{-1}$) Given a WS1S formula, the free variable set is $Y'$ and assume they are encoded from UABE variable set $Y$. A value decoding function is a function $\delta^{-1}$ which translate an interpretation $\sigma_V$ (on $Y'$) to a valuation $V$ (on $Y$) such that:

- For variable $i \in Y \cap \mathcal{V}(\mathbb{N})$: assume $i$ is encoded by $i'$. Then it must hold that $V(i) = \sigma_V(i')$.
- For variable $x \in Y \cap \mathcal{V}(\mathbb{Z}_n)$, assume $x$ is encoded by $S_x$. Then it must hold that $\forall 0 \leq d < n.(V(x)^{[d]} = 1 \leftrightarrow d \in \sigma_V(S_x))$.
- For variable $a \in Y \cap \mathcal{V}(\mathbb{A})$: assume $a$ is encoded by $|a|, a^{(0)}, \ldots, a^{(d-1)}$. Then it must hold:

$$V(|a|) = \sigma_V(|a|) \wedge \forall 0 \leq d < n, 0 \leq i < V(|a|).\left(V(a[i])^{[d]} = 1\right) \leftrightarrow (i \in \sigma_V(a^{(d)}))$$

Note that $\delta$ is an injection but $\delta^{-1}$ is not. If some $S_x$ contains some elements that are no less than $n$, they will be ignored when decoding. So $\delta^{-1}$ is not the inverse function of $\delta$. However, it is correct that

**Lemma 5** *For all valuation V,* $\delta^{-1}(\delta(V)) = V$.

*Proof* This lemma is correct by definition. □

**Theorem 3** *Given any UABE formula $f$ and any interpretation $\sigma_V$ such that $\sigma_V \models_{\mathsf{WS1S}} \Theta$, then it holds that:*

$$\delta^{-1}(\sigma_V) \models_{\mathsf{UABE}} f \iff \sigma_V \models_{\mathsf{WS1S}} \Delta(f)$$

The proof is similar to that for Theorem 2. We do not repeat it here.

Theorems 2 and 3 actually show the fact that for each UABE formula $f$, there is a *correspondence* between valuations for $f$ and interpretations for $\Delta(f)$ that satisfies $\Theta$.

**Theorem 4** *Given any UABE formula $f$, $f$ is satisfiable if and only if $\Theta \wedge \Delta(f)$ is satisfiable, $f$ is valid if and only if $\Theta \rightarrow \Delta(f)$ is valid.*

*Proof* We first prove that $f$ is satisfiable if and only if $\Theta \wedge \Delta(f)$ is. If $f$ is satisfiable, then there exists some $V$ such that $V \models_{\mathsf{UABE}} f$. By Theorem 2, we know $\sigma_A \circ \delta(V) \models_{\mathsf{WS1S}} \Theta \wedge \Delta(f)$. For the other direction, if there is some $\sigma_V$ such that $\sigma_V \models_{\mathsf{WS1S}} \Theta \wedge \Delta(f)$, then it is obvious that $\sigma_V \models_{\mathsf{WS1S}} \Theta$. We know by Theorem 3 that $\delta^{-1}(\sigma_V) \models_{\mathsf{UABE}} f$.

For the other half of this theorem, $f$ is valid $\iff \neg f$ is unsatisfiable $\iff \Theta \wedge \neg f$ is unsatisfiable $\iff \neg(\Theta \rightarrow f)$ is unsatisfiable $\iff \Theta \rightarrow f$ is valid. □

**Theorem 5** *The satisfiability problem for UABE is decidable.*

*Proof* The function $\Delta(\cdot)$ is a linear translation procedure. Because the satisfiability problem for WS1S is decidable, so is UABE. □

If the size of a formula is measured by the number of nodes in the syntax tree, the translated formula is at most $O(n)$ times larger than the original one. Such conclusion can be obtained by analyzing each translation rule.

An observation is that the element domain of UABE could be generalized. For instance, negative numbers can be encoded in $\mathbb{Z}_n$ using a sign bit. Those predicates $P_=$, $P_<$ and $P_+$ have to be refined accordingly. In fact, the essential requirement for the element domain is that it can be encoded in WS1S.

## 5 Extensions

In this section, we consider extending UABE to enrich its expressiveness. Currently, there are two major restrictions for UABE, one is that the array elements are bounded and the other is that addition is not allowed on two index variables. So the natural extensions are:

- Let the domain of the element theory be unbounded.
- Allow addition of index variables such as $k = i + j$.

In both cases, we prove their satisfiability problem are undecidable by reducing the Hilbert's tenth problem [21] to them.

*Hlibert's Tenth Problem* The Hilbert's Tenth Problem is a well-known undecidable problem. In short, it is the problem of deciding whether the following equation has any (non-negative) integer solutions:

$$p(x_1, x_2, \ldots, x_n) = 0$$

here $p$ is a polynomial of integer coefficients.

To reduce the Hilbert's Tenth Problem to the extended array theory, what we need to do is to define, in the extended theory, a formula $\varphi_+(k, i, j)$ which is satisfiable if and only if $k = i + j$, and a formula $\varphi_\times(k, i, j)$ which is satisfiable if and only if $k = i \times j$. Here, $k, i, j$ are non-negative integer variables. Furthermore, we can assume $k, i, j$ are strictly positive because the definition is trivial when one of them is 0.

## 5.1 Extend Array Elements to $\mathbb{N}$

We call this extended theory UAUE (Unbounded Array with Unbounded Elements). Knowing that $i, j, k$ are positive, we can define $\varphi_+(k, i, j)$ and $\varphi_\times(k, i, j)$ as follows:

$$\begin{aligned}
\varphi_+(k, i, j) \equiv\ & \exists a.\ |a| > j \wedge a[0] = i \wedge \forall p.\ (p < j \rightarrow a[p+1] = a[p] + 1) \wedge k = a[j] \\
\varphi_\times(k, i, j) \equiv\ & \exists a, b\,.\, a[0] = i \wedge b[0] = 0 \wedge a[k] = 0 \wedge b[k] = 0 \\
& \wedge \forall p.\ (b[p] > 0 \rightarrow a[p+1] = a[p] \wedge b[p+1] = b[p] - 1) \\
& \wedge \forall p.\ (a[p] > 0 \wedge b[p] = 0 \rightarrow a[p+1] = a[p] - 1 \wedge b[p] = j - 1) \\
& \wedge \forall p.\ (a[p] = 0 \wedge b[p] = 0 \rightarrow k \leq p)
\end{aligned}$$

**Lemma 6** *$\varphi_+(k, i, j)$ is satisfiable if and only if $k = i + j$.*

*Proof* $\varphi_+(k, i, j)$ is constructed in such a way that $a[p] = i + p$ holds for all $p \leq j$. So $a[j] = i + j$. If $\varphi_+(k, i, j)$ is satisfiable, then $k = a[j] = i + j$, and vice versa. □

**Lemma 7** *$\varphi_\times(k, i, j)$ is satisfiable if and only if $k = i \times j$.*

*Proof* $\varphi_\times(k, i, j)$ is constructed in such a way that $a[p] \times j + b[p] + p = i \times j$ holds for all $0 \leq p \leq i \times j$. Initially, $a[0] = i$ and $b[0] = 0$. Then the value of $a[p] \times i + b[p]$ decreases by 1 each step when $p$ increases. $k$ is the minimal index such that $a[k] = b[k] = 0$. From the equation we know $0 \times j + 0 + p = p = i \times j$. Therefore, when $\varphi_\times(k, i, j)$ is satisfiable, we have $k = i \times j$. Vice versa. □

**Theorem 6** *The satisfiability problem of UAUE is undecidable.*

*Proof* An equation that contains polynomial of variables can be formulated with the help of $\varphi_+$ and $\varphi_\times$. Thus the Hilbert's Tenth Problem is reducible to the satisfiability problem of UAUE. Since the former one is undecidable, so is the latter one. □

### 5.2 Allow Additions on $\mathbb{N}$ Variables

We call this extended theory UABE$^+$. If we extend the theory of index to allow additions, $\varphi_+(k, i, j)$ can be defined directly as $k = i + j$. It remains to define $\varphi_\times(k, i, j)$. At first, we assume $i, j$ are both no less than 2.

The proof in the previous section can not be applied because it relies on the fact that elements of an array can be arbitrarily large, but it is not the case for UABE$^+$. However, we can assume the domain of array element contains at least two values, say $\{0, 1\}$. Then an array $a$ can be treated as a finite set of integers which consists of all the indices where the corresponding array element is 1. Use $\hat{a}$ to denote the set represented by the array $a$. i.e: $\hat{a} = \{i \in \mathbb{N} \mid a[i] = 1\}$.

In this section, we also use:

- $[x, y]$ to denote the least common multiple of $x$ and $y$;
- $\langle x \rangle$ to denote the set $\{\alpha \cdot x \mid \alpha \in \mathbb{N}\}$, i.e, all multiples of $x$;
- $\langle x, y \rangle$ to denote the set $\{\alpha \cdot [x, y] \mid \alpha \in \mathbb{N}\}$, i.e, all multiples of $[x, y]$;

It is obvious that: $\forall x, y. \langle x, y \rangle = \langle [x, y] \rangle = \langle x \rangle \cap \langle y \rangle$

A finite set $P$ is a *prefix* of some set $Q$ (finite or infinite) if there is an integer $r$ such that $P = \{0, 1, 2, \ldots, r\} \cap Q$. denoted by $P \sqsubset Q$. If $P$ is a set, $c$ is a non-negative integer, then $P + c = \{\alpha + c \mid \alpha \in P\}$ is the set obtained by adding $c$ to each element in $P$.

Use $\Pi(i, j)$ to denote the set $= \langle i, j \rangle \cap (\langle i - 1, j - 1 \rangle + i + j - 1)$. The idea behind the construction is expressed as the following lemma:

**Lemma 8** *If $i, j \geq 2$, then $i \times j$ is the minimum element in $\Pi(i, j)$.*

*Proof* Firstly, $i \cdot j - i - j + 1 = (i - 1)(j - 1)$ implies $i \cdot j - i - j + 1 \in \langle i - 1, j - 1 \rangle$. Thus $i \cdot j \in (\langle i - 1, j - 1 \rangle + i + j - 1)$ and $i \cdot j \in \Pi(i, j)$.

We prove that $i \cdot j$ is the minimal element by contradiction. Assume there is an integer $p \in \Pi(i, j)$ and $p < i \cdot j$. Then $p \in \langle i, j \rangle$ and $p - i - j + 1 \in \langle i - 1, j - 1 \rangle$ both hold, which implies $i | p$, $j | p$, $(i - 1) | (p - i - j + 1)$ and $(j - 1) | (p - i - j + 1)$.

Suppose $[i, j] = i \cdot j / b$ where $b$ is the greatest common divisor of $i$ and $j$. then $p = t[i, j] = t \cdot (i \cdot j / b)$ where $1 \leq t < b$. Since $(i - 1) | (p - i - j + 1)$, we know:

$$(i-1) | (p-j) \iff (i-1) | \left( \frac{t \cdot i \cdot j}{b} - j \right)$$

$$\iff (i-1) | \left( ((i-1) + 1) \frac{t \cdot j}{b} - j \right) \iff (i-1) | \left( (i-1) \frac{t \cdot j}{b} + \frac{t \cdot j}{b} - j \right)$$

$$\iff (i-1) | \left( \frac{t \cdot j}{b} - j \right) \iff (i-1) | (t-b)\, j/b$$

Symmetrically, $(j - 1) | (t - b) i / b$. Hence $(i - 1)(j - 1) | (b - t)^2 \times j/b \times i/b$.

We assert that $(i - 1) > (b - t) \times i/b$, it suffices to show $t \cdot i > b$: since $p \in \Pi(i, j)$, it is a necessary condition that $p \geq i + j - 1$. i.e: $(t \cdot i \cdot j/b \geq i + j - 1) \Rightarrow (t \cdot i \cdot j/b > j) \Rightarrow (t \cdot i > b)$. Symmetrically, we know $(j - 1) > (b - t) \times j/b$. Hence $(i - 1)(j - 1) > (b - t)^2 \times j/b \times i/b$ holds. Combined with the fact that $(i - 1)(j - 1) | (b - t)^2 \times j/b \times i/b$, we know $(b - t)^2 \times j/b \times i/b = 0$ must hold. But this is impossible because $t < b$ and $i, j, b, t$ are all positive integers. Contradiction. □

Following the idea presented in the previous lemma, we are able to construct $\varphi_\times(k, i, j)$. But the problem is, sets such as $\langle i, j\rangle$ are infinite, while arrays are finite in UABE$^+$. Good news is the finite (but arbitrarily large) prefixes of these infinite sets are sufficient to define $\varphi_\times(k, i, j)$. We use a variable $\gamma$ to bound the size of arrays and quantify $\gamma$ with an existential quantifier. For sufficiently large $\gamma$ (larger than $i \times j$), everything works through. Detailed steps are:

- construct a set $\hat{m}_i \sqsubset \langle i\rangle$, by asserting $F_1$:

$$(m_i[0] =_n 1) \wedge (\forall p.(0 < p < i) \rightarrow m_i[p] =_n 0) \wedge (\forall p.p < \gamma \rightarrow m_i[p] =_n m_i[p + i])$$

- construct a set $\hat{m}_j \sqsubset \langle j\rangle$ in the same way, call it $F_2$;
- construct a set $\hat{m}_{i,j} \sqsubset \hat{m}_i \cap \hat{m}_j$, i.e: $\hat{m}_{i,j} \sqsubset \langle i, j\rangle$ by asserting $F_3$:

$$\forall p < \gamma.((m_{i,j}[p] =_n 1) \leftrightarrow (m_i[p] =_n 1 \wedge m_j[p] =_n 1))$$

- construct $\hat{m}_{i-1,j-1}$ in the same way, call it $F_4$;
- shift every element in $\hat{m}_{i-1,j-1}$ by $i + j - 1$, result in $\hat{m}'_{i-1,j-1}$ by asserting $F_5$:

$$\begin{aligned} &\forall p.((p < i + j - 1) \rightarrow m'_{i-1,j-1}[p] =_n 0) \\ \wedge\ &\forall p < \gamma.m_{i-1,j-1}[p] =_n m'_{i-1,j-1}[p + i + j - 1] \end{aligned}$$

- find the minimal number in the intersection of $m_{i,j}$ and $m'_{i-1,j-1}$, say it $k$, by asserting $F_6$:

$$(m_{i,j}[k] = 1 \wedge m'_{i-1,j-1}[k] =_n 1) \wedge (\forall p.(m_{i,j}[p] =_n 1 \wedge m'_{i-1,j-1}[p] = 1) \rightarrow k \leq p)$$

We conjunct all these $F_i$ and quantify free variables with an existential quantifier, let:

$$\psi(k, i, j) = \exists \gamma, m_i, m_j, m_{i,j}, m_{i-1}, m_{j-1}, m_{i-1,j-1}, m'_{i-1,j-1}.(\wedge_{i=1}^{6} F_i)$$

By Lemma 8, we know when $\gamma$ is sufficiently large, such $k$ will exist in the intersection of $m_{i,j}$ and $m'_{i-1,j-1}$. So, $\psi(k, i, j)$ is satisfiable if and only if $k = i \times j$.

It is trivial when one of $i, j$ is less than 2. Hence $\varphi_\times(k, i, j)$ is defined as:

$$\begin{aligned} &((i = 0 \vee j = 0) \rightarrow k = 0) \\ \wedge\ &(i = 1 \rightarrow k = j) \wedge (j = 1 \rightarrow k = i) \\ \wedge\ &(i \geq 2 \wedge j \geq 2 \rightarrow \psi(k, i, j)) \end{aligned}$$

**Theorem 7** *The satisfiability for UABE$^+$ is undecidable.*

We mention the fact that "Presburger Arithmetic with predicate is undecidable" is already proved in [15]. However it can not be used directly because arrays are finite in UABE$^+$ and finite arrays can not be used to encode predicates over the infinite domain $\mathbb{N}$.

## 6 Implementation and Experiments

Implementation issues are discussed in this section. Then we conduct the experiments on some array formula examples and in program verification. All experiments are carried out on a PC with 2.53 GHz CPU and 2 GB memory.

### 6.1 Implementation

A decision procedure for UABE is implemented on top of MONA (version 1.4). It acts following the steps below:

1. Parse the input formula $f_0$.
2. Rewrite $f_0$ into $f$, where $f$ is logically equivalent to $f_0$ and is in the minimal syntax.
3. Apply the translation rules to $f$ until saturation. The result is $\Delta(f)$.
4. Generate $\Theta \wedge \Delta(f)$ (and $\Theta \rightarrow \Delta(f)$) and pass it to MONA.
5. If MONA returns "satisfiable" or "valid", report the satisfying-example; otherwise report "unsatisfiable" as well as the counter-example.

For example, the following formula states that an array $a$ is *sorted*.

$$\forall i, j : \mathbb{N}.\ (i < j < |a| \rightarrow a[i] \leq a[j])$$

Assume $n = 8$, to check its satisfiability, the above formula is first translated to the minimal syntax: $\forall i, j : \mathbb{N}.\ ((i < j \wedge j < |a|) \rightarrow a[i] \leq a[j])$, and then to the WS1S formula $\Theta \wedge \Delta(f)$ (Fig. 1). In the WS1S formula, the first few lines are the definition of $\Theta$. Then follows the predicates such as $P_<$. The last part is the formula $\Theta \wedge \Delta(f)$. We can check its satisfiability by calling MONA. The verification result is satisfiable

```
# definition for A~[i] = i
pred Theta() = ...;

# definition for P<(x, y)
pred Plt(var0 x7, x6, ..., x0, y7, y6, ..., y0)
= (~x7 & y7) |
  (x7 <=> y7) & ((~x6 & y6) | (x6 <=> y6) & ((~x5 & y5) |
  ...
  (x2 <=> y2) & ((~x1 & y1) | (x1 <=> y1) & ((~x0 & y0))))))));

# variables
var2 a_7, a_6, ..., a_0;
var1 len_a;

# Theta /\ Delta(f)
Theta &
(all1 i: (all1 j: (((i < j) & (j < len_a)) => (
    (Ple((i in a_7), ..., (i in a_0), (j in a_7), ..., (j in a_0))) &
    (i < len_a) & (j < len_a))))
);
```

**Fig. 1** An example of translated WS1S formula

and an example is returned. A valuation for the array $a$ can be obtained by analyzing the returned example.

### 6.2 Verify Array Properties

In order to test our decision procedure, we collected a few array properties which are frequently considered when writing programs. For example, some asserts that an array is sorted, or any subsequence of a sorted array is still sorted, or an array is a Fibonacci sequence.

These properties are formulated in UABE. Then our tool is applied to solve their satisfiability (or validity). The experimental results are shown in Table 5, where the $n$ column gives the number of bits used to encode an array element, the *UABE* and *WS1S* columns list the size of formulas (measured in the number of nodes of the syntax tree) respectively, the *Status* column gives the verification results (either satisfiable, valid or unsatisfiable), and the *Time* and *Mem* columns show the resource consumption for solving these problems. Time is measured in seconds and memory is measured in megabytes. The last column is a brief description of the array property.

As shown in Table 5, all instances can be solved in no more than 20 seconds. On the contrary, the memory consumption is large, ranging from 1 M to 1.4 G. As mentioned above, our solver is built on top of Mona. Mona employs automaton and BDD to solve WS1S formulas. The BDD size can grow non-elementarily with respect to the number of quantifier alternations.

**Table 5** Experiment on array formulas

| $n$ | $f$ | UABE | WS1S | Status | Time | Mem | Note |
|---|---|---|---|---|---|---|---|
| 8 | $f_1$ | 5 | 35 | Sat | 0.04 s | 1 M | Normal equality |
| 8 | $f_2$ | 22 | 180 | Sat | 0.31 s | 1 M | Equality with addition |
| 4 | $f_3$ | 19 | 166 | Sat | 0.12 s | 1 M | Array read |
| 7 | $f_3$ | 19 | 1058 | Sat | 10.4 s | 16 M | Array read |
| 4 | $f_4$ | 45 | 280 | Sat | 0.12 s | 1 M | Addition on index |
| 8 | $f_5$ | 22 | 145 | Sat | 0.35 s | 1 M | Monotonically increasing array |
| 8 | $f_6$ | 29 | 220 | Sat | 9.07 s | 533 M | Fibonacci array |
| 4 | $f_7$ | 23 | 60 | Sat | 2.12 s | 136 M | Sorted property |
| 4 | $f_8$ | 48 | 114 | Valid | 18.03 s | 1416 M | Sorted implies distinct |
| 4 | $f_9$ | 37 | 165 | Sat | 15.39 s | 671 M | Periodical array |
| 4 | $f_{10}$ | 29 | 66 | Sat | 18.09 s | 1343 M | Distinct array |
| 4 | $f_{11}$ | 32 | 69 | Sat | 10.09 s | 656 M | Partitioned array |
| 8 | $f_{13}$ | 25 | 300 | Sat | 0.06 s | 1 M | Array write |
| 8 | $f_{15}$ | 16 | 74 | Valid | 0.03 s | 1 M | Unbounded array |
| 8 | $f_{16}$ | 54 | 345 | Sat | 0.10 s | 1 M | Quantifier alternation |
| 4 | $f_{17}$ | 20 | 84 | Sat | 9.06 s | 200 M | Miscellaneous array property |
| 6 | $f_{17}$ | 20 | 116 | Sat | 12.19 s | 330 M | Miscellaneous array property |
| 7 | $f_{17}$ | 20 | 132 | Sat | 17.11 s | 340 M | Miscellaneous array property |
| 8 | $f_{18}$ | 80 | 617 | Sat | 0.70 s | 1 M | Count elements in an array |
| 7 | $f_{19}$ | 53 | 385 | Sat | 2.37 s | 147 M | Find the maximal number |
| 4 | $f_{20}$ | 63 | 129 | Valid | 3.27 s | 259 M | Sorted implies sub-array sorted |
| 8 | $f_{21}$ | 53 | 298 | Valid | 4.78 s | 199 M | Array axiom |

### 6.3 Applications in Program Verification

In this subsection, we report the results in program verification. Five program instances are considered:

- *array shift*: A program that right-shifts an array with a positive size. Desired property is that after shifting, each element is in the expected position.
- *find max*: A simple program that traverse an array and find the maximum value. The property to verify is that the returned value does appear in the array and it is indeed the maximum value.
- *binary search*: Implementation of the typical binary search algorithm. The property to verify is that if the desired element exists in the array, the program will eventually locate it.
- *bucket counting*: The sample program given at the beginning of this paper. The property to verify is that for each bucket $v$ there should be at least one element in the array holding this value.
- *selection sort*: Implementation of selection sort and we verify that it returns a sorted array.

The verification results are given in Table 6. For each program instance, we generated several verification conditions. The correctness of the program instance is ensured by the validity of all verification conditions. Notice that several verification conditions are not in the decidable fragments of other decision procedures.

In the following, we elaborate the details for verifying *bucket counting* and *selection sort*. The details for *array shift*, *find max* and *binary search* are omitted.

#### *6.3.1 Bucket Counting*

This example is introduced at the beginning of this paper. $a$ is an array whose elements range over [0..255]. The number of different elements in $a$ needs to be counted. $b$ is an array used as indicators such that $b[i] = 1$ iff $i$ appears in $a$. The key step is to set $b[a[i]] = 1$ for all indices $i \in [0, |a| - 1]$. For convenience, the pseudo-code is rewritten as a `while` loop as follows:

$$
\begin{array}{l}
i \leftarrow 0; \\
\texttt{while}\ i < |a| \\
\quad b[a[i]] \leftarrow 1; \\
\quad i \leftarrow i + 1; \\
\texttt{endwhile}
\end{array}
$$

If we can guarantee before this loop, that $|b| \geq 256 \wedge \forall 0 \leq v < |b|.\ b[v] = 0$, then after this loop, a property should hold that

$$\forall v \in [0, 255].\ (b[v] = 1 \leftrightarrow (\exists i.a[i] = v))$$

We verify this property in Hoare-Logic [17]. We also assume the program is already annotated and verification conditions are generated. Technically, we need to verify that all verification conditions are tautologies so that the desired property hold.

**Table 6** Experiment on program verification

| Program | $n$ | $f$ | UABE | WS1S | Status | Time | Mem |
|---|---|---|---|---|---|---|---|
| Array shift | 8 | $vc_1$ | 45 | 181 | Valid | 0.03 s | 1 M |
| Array shift | 8 | $vc_2$ | 95 | 383 | Valid | 0.03 s | 1 M |
| Array shift | 8 | $vc_3$ | 112 | 578 | Valid | 13.3 s | 619 M |
| Array shift | 8 | $vc_4$ | 74 | 310 | Valid | 5.17 s | 157 M |
| Find max | 4 | $vc_1$ | 40 | 104 | Valid | 0.03 s | 1 M |
| Find max | 4 | $vc_2$ | 87 | 210 | Valid | 0.03 s | 1 M |
| Find max | 4 | $vc_3$ | 93 | 235 | Valid | 17.3 s | 170 M |
| Find max | 4 | $vc_4$ | 92 | 231 | Valid | 24.3 s | 768 M |
| Find max | 4 | $vc_5$ | 74 | 163 | Valid | 21.09 s | 500 M |
| Binary search | 4 | $vc_1$ | 65 | 177 | Valid | 2.21 s | 100 M |
| Binary search | 4 | $vc_2$ | 39 | 147 | Valid | 0.06 s | 1 M |
| Binary search | 4 | $vc_3$ | 50 | 183 | Valid | 0.06 s | 1 M |
| Binary search | 4 | $vc_4$ | 96 | 308 | Valid | 2.35 s | 118 M |
| Binary search | 4 | $vc_5$ | 30 | 113 | Valid | 0.06 s | 18 M |
| Bucket counting | 2 | $vc_1$ | 43 | 114 | Valid | 0.09 s | 1 M |
| Bucket counting | 2 | $vc_2$ | 85 | 239 | Valid | 0.49 s | 50 M |
| Bucket counting | 2 | $vc_3$ | 65 | 162 | Valid | 0.01 s | 1 M |
| Bucket counting | 2 | $vc_4$ | 61 | 294 | Valid | 0.29 s | 2 M |
| Selection sort | 3 | $vc_1$ | 57 | 109 | Valid | 0.0 s | 1 M |
| Selection sort | 3 | $vc_2$ | 127 | 225 | Valid | 0.0 s | 1 M |
| Selection sort | 3 | $vc_3$ | 90 | 165 | Valid | 0.03 s | 1 M |
| Selection sort | 3 | $vc_4$ | 187 | 329 | Valid | 0.04 s | 1 M |
| Selection sort | 3 | $vc_5$ | 259 | 695 | Valid | 0.1 s | 1 M |
| Selection sort | 3 | $vc_6$ | 259 | 467 | Valid | 0.1 s | 1 M |
| Selection sort | 3 | $vc_7$ | 259 | 467 | Valid | 0.07 s | 1 M |
| Selection sort | 3 | $vc_8$ | 239 | 425 | Valid | 0.07 s | 1 M |
| Selection sort | 3 | $vc_9$ | 203 | 501 | Valid | 0.17 s | 1 M |

An observation is that "$b\,[a[i]] \leftarrow 1$" is actually an array write expression "$b \leftarrow \mathsf{write}(b\,, a[i], 1)$". Moreover, the range of elements in $a$ is restricted to $[0, 255]$ if they are encoded by 8-bit integers. The annotated program is given as in Fig. 2. Among these specifications, $P_2$ is a loop invariant. The verification conditions generated are listed below.

$$vc_1 \equiv P_1 \rightarrow P_2[i/0]$$

$$vc_2 \equiv P_2 \wedge (i < |a|) \rightarrow P_3[b\,/\mathsf{write}(b\,, a[i], 1)]$$

$$vc_3 \equiv P_3 \rightarrow P_2[i/i+1]$$

$$vc_4 \equiv P_2 \wedge \neg(i < |a|) \rightarrow P_4$$

Here, $P[x/e]$ is the formula obtained by replacing free occurrences of $x$ with $e$.

It is obvious that all $vc_i$ contain quantifier alternation. They are not in the decidable fragment of other known theories. However, all of them are in the decidable fragment of UABE. Despite of the large time complexity, it is possible to automatically verify that these $vc_i$ are indeed tautologies using our decision procedure. Therefore, the property $P_4$ holds after this program as desired.

**Fig. 2** Annotated bucket counting program

$\{P_1 \equiv\ |b| \geq 255 \wedge \forall 0 \leq v < |b|.\ (b[v] = 0)\}$
$i \leftarrow 0;$
$\{\textbf{inv: } P_2 \equiv\ \forall x : \mathbb{N}.\ (b[x] = 1 \leftrightarrow \exists p.(p < i \wedge a[p] = x))\}$
`while` $i < |a|$ :
  $b \leftarrow \mathsf{write}(b, a[i], 1);$
  $\{P_3 \equiv\ \forall x : \mathbb{N}.\ (b[x] = 1 \leftrightarrow \exists p.(p < i + 1 \wedge a[p] = x))\}$
  $i \leftarrow i + 1;$
`endwhile`
$\{P_4 \equiv\ \forall v : \mathbb{N}.\ (b[v] = 1 \leftrightarrow \exists p.a[p] = v)\}$

### *6.3.2 Selection Sort*

Another example is selection sort. Given an array $a$, the selection sort algorithm runs for $|a|$ iterations. In the $i$-th iteration, the smallest element in $\{a[j] \mid (i-1) \leq j < |a|\}$ is swapped with $a[i]$. After $|a|$ iterations, the array is sorted in ascending order. We assume that the program is annotated as in Fig. 3. In the specifications, a few "predicates" such as min, sorted, partitioned are used:

– $\mathsf{sorted}(a, l, r)$ formulates the proposition that elements in the interval $[l, r]$ are sorted in ascending order.

$$\mathsf{sorted}(a, l, r) \equiv \forall l \leq i < j \leq r.a[i] \leq a[j]$$

– $\mathsf{partitioned}(a, l, p, r)$ formulates the proposition that a segment (index interval $[l, r]$) in array $a$ is partitioned at the index $p$.

$$\mathsf{partitioned}(a, l, p, r) \equiv \forall i, j.(l \leq i < p \wedge p \leq j \leq r \rightarrow a[i] \leq a[j])$$

$\{P_1 \equiv |a| > 0\}$
$i \leftarrow 0;$
$\{\textbf{inv} : P_2 \equiv 0 \leq i \wedge \mathsf{sorted}(a, 0, i-1) \wedge \mathsf{partitioned}(a, 0, i, |a|-1)\}$
`while`$(i < |a|)$ :
  $\{P_3 \equiv 0 \leq i < |a| \wedge \mathsf{sorted}(a, 0, i-1) \wedge \mathsf{partitioned}(a, 0, i, |a|-1)\}$
  $k \leftarrow i;$
  $j \leftarrow i + 1;$
  $\left\{\textbf{inv} : P_4 \equiv \begin{array}{l} i < j \wedge i \leq k \wedge 0 \leq i, k < |a| \wedge 0 \leq j \leq |a| \\ \wedge \mathsf{min}(a[k], a, i, j-1) \wedge \mathsf{sorted}(a, 0, i-1) \wedge \mathsf{partitioned}(a, 0, i, |a|-1) \end{array}\right\}$
  `while`$(j < |a|)$ :
    $\left\{P_5 \equiv \begin{array}{l} i < j \wedge i \leq k \wedge 0 \leq i, k < |a| \wedge 0 \leq j < |a| \\ \wedge \mathsf{min}(a[k], a, i, j-1) \wedge \mathsf{sorted}(a, 0, i-1) \wedge \mathsf{partitioned}(a, 0, i, |a|-1) \end{array}\right\}$
    `if`$(a[j] < a[k])$ :
      $k \leftarrow j;$
    `endif`
    $j \leftarrow j + 1;$
  `endwhile`
  $\left\{P_6 \equiv \begin{array}{l} 0 \leq i \leq k < |a| \\ \wedge \mathsf{min}(a[k], a, i, |a|-1) \wedge \mathsf{sorted}(a, 0, i-1) \wedge \mathsf{partitioned}(a, 0, i, |a|-1) \end{array}\right\}$
  $a_k \leftarrow a[k];$
  $a_i \leftarrow a[i];$
  $a \leftarrow \mathsf{write}(a, i, a_k);$
  $a \leftarrow \mathsf{write}(a, k, a_i);$
  $i \leftarrow i + 1;$
`endwhile`
$\{P_7 \equiv \mathsf{sorted}(a, 0, |a|-1)\}$

**Fig. 3** Annotated selection sort program

– $\mathsf{min}(x, a, l, r)$ formulates the proposition that $x$ is the minimal value in a segment (index interval $[l, r]$) of the array $a$.

$$\mathsf{min}(x, a, l, r) \equiv (\exists l \leq i \leq r.x = a[i]) \wedge (\forall l \leq i \leq r.a[i] \leq x)$$

For this example, verification conditions are:

$$vc_1 \equiv P_1 \rightarrow P_2[i/0]$$

$$vc_2 \equiv P_2 \wedge (i < |a|) \rightarrow P_3$$

$$vc_3 \equiv P_2 \wedge \neg(i < |a|) \rightarrow P_7$$

$$vc_4 \equiv P_3 \rightarrow P_4[j/i+1][k/i]$$

$$vc_5 \equiv P_4 \wedge (j < |a|) \rightarrow P_5$$

$$vc_6 \equiv P_5 \wedge (a[j] < a[k]) \rightarrow P_4[j/j+1][k/j]$$

$$vc_7 \equiv P_5 \wedge \neg(a[j] < a[k]) \rightarrow P_4[j/j+1]$$

$$vc_8 \equiv P_4 \wedge \neg(j < |a|) \rightarrow P_6$$

$$vc_9 \equiv P_6 \rightarrow P_2[i/i+1][a/\mathsf{write}(a, k, a_i)][a/\mathsf{write}(a, i, a_k)][a_i/a[i]][a_k/a[k]]$$

Notice that these are non-trivial array formulas. Good news is our decision procedure still managed to verify all of them. Thus, we know the sorted property holds.

During the verification, some properties caught our attention. For instance, "a sub-segment of a sorted segment is still sorted"

$$\forall l_1, r_1, l_2, r_2.\, l_1 \leq l_2 \wedge r_2 \leq r_1 \rightarrow \mathsf{sorted}(a, l_1, r_1) \rightarrow \mathsf{sorted}(a, l_2, r_2)$$

This property has complex Boolean structure and quantifiers (included in the sorted predicate). It is good to see that they are verified by our decision procedure.

## 7 Related Works

Quantifier-free array formulas have been well studied. In [25], a non-extensional array theory is discussed and its decision procedure is given based on the congruence closure algorithm. An extensional array theory is considered in [26]. Its satisfiability is also checked by a variant of congruence closure algorithm. In [18], a reduction approach for array theory is proposed. Array formulas are converted to formulas in the theory of equality. Other data structures such as lists, sets and multisets are also considered. Combinatory array logic is considered in [7]. Sets and bags can be formulated in that way. Its decision procedure is given as inference rules. In [13], frugality of axiom instantiation for array theory is discussed. Arrays are treated as updatable functions. While it is common to eliminate the write function in most works, the authors present a write based approach and eliminate read on the other hand. In [12], a rewriting based decision procedure is proposed, in which array length (referred as *dimension* in that paper) could be used as an attribute of an array. Predicates such as *sorted* could be interpreted and handled in their decision procedure. Similar work could be found in [27] where the *PERM* predicate

is explicitly considered for practical interest. While it is almost standard to model arrays as uninterpreted functions, there are also eager approaches that employ SAT solving for array theories. Such works could be found in [4, 9] and [5]. All above works consider quantifier-free array theories.

In [3], a more expressive logic is presented. The authors defined a $\exists^*\forall^*$-fragment of array theory. The index theory is alike to Presburger arithmetic, but addition is only allowed on the existentially quantified index variables. The authors present a mechanism to instantiate universal quantifiers by replacing the quantified index variables with each term in the index set. Then arrays are treated as uninterpreted functions. It is shown that allowing nested reads or general Presburger arithmetic over universally quantified index variables (even just $i + 1$) induces undecidability. Although predicates such as sorted can be defined naturally, many verification conditions listed in the experiments can not be handled by their approach.

A complete quantifier instantiation algorithm is presented in [10]. Its array theory subsumes that in [3]. For instance, nested reads and offsets on indices are allowed. For certain subclasses of their array formulas, model-based quantifier instantiation is used to decide the satisfiability. Their algorithm is complete as long as the constraints set introduced by the formula has a finite solution. For certain cases, such as those contain offsets on array indices, their procedure will result in an infinite set of instantiations.

In [28], quantified bit-vector formulas (QBVF) are considered. Arrays could be formulated as uninterpreted functions which are supported in QBVF. Both indices and elements are bit-vectors. QBVF formulas could be reduced to equisatisfiable EPR formulas whose Herbrand universe is always finite, so the satisfiability of QBVF is decidable. The authors present a group of rewriting rules for preprocessing and use template based model finding to check quantifiers. For their array theory to be decidable, arrays should have bounded length.

An automata based approach is presented in [14]. The authors define an array theory SIL which allows formulas like $\forall i.\phi(i) \rightarrow \gamma(i)$ where $i$ is the only index variable in $\gamma(i)$ and disjunction is forbidden in $\gamma(i)$. The same as in [3], quantifiers can be used only in the form of $\exists^*\forall^*$. An SIL formula is translated into a flat counter automaton, then the satisfiability is checked by testing if the language of automaton is empty. Furthermore, in [2], they encode pieces of program into counter automata. In SIL, there are some unnatural restrictions on the structure of formulas. Nested reads are not allowed, either.

A theory of integer sequences is proposed in [8]. The operations of concatenating two sequences are supported. Universal quantifiers can be used at the beginning of the formula. The decidability result is obtained by reducing to the theory of concatenation. However, the $\forall^*\exists^*$ and $\exists^*\forall^*$ fragments are undecidable either. Moreover, it does not support explicitly access by indices such as $a[i]$. That is a big inconvenience when writing array formulas.

There are incomplete algorithms handling array formulas with arbitrary quantifiers. Those algorithm may not terminate for all inputs. References could be found in [11] and [24].

Table 7 is a summary of the above related works. $\mathcal{T}_I$ and $\mathcal{T}_E$ are theories of index and element respectively. $\mathcal{P}$ is the Presburger Arithmetic and "bv" is the theory of bit-vector. A star "$*$" means the theory is parameterized. The right half of table describes features for each individual theory. The "$\exists$, $\forall$" column represents

**Table 7** Summary of related works

| Reference | $\mathcal{T}_I$ | $\mathcal{T}_E$ | $\exists, \forall$ | $a[a[\cdot]]$ | $\lvert\cdot\rvert$ | ext | dim |
|---|---|---|---|---|---|---|---|
| Nelson [25] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\times$ | $n$ |
| Stump et al. [26] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| Kapur and Zarba [18] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| de Moura and Bjorner [7] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| Goel et al. [13] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| Bofill et al. [1] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| Suzuki and Jefferson [27] | $\mathcal{P}$ | $\mathcal{P}$ | $\times$ | $\surd$ | $\times$ | $\times$ | 1 |
| Ghilardi et al. [12] | $\mathcal{P}$ | $*$ | $\times$ | $\surd$ | $\surd$ | $\surd$ | $n$ |
| Ganesh and Dill [9] | bv | bv | $\times$ | $\surd$ | $\times$ | $\times$ | 1 |
| Brummayer and Biere [5] | bv | bv | $\times$ | $\surd$ | $\times$ | $\surd$ | 1 |
| Brummayer and Biere [4] | $*$ | $*$ | $\times$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| Bradley et al. [3] | $\mathcal{P}$ | $*$ | $\odot$ | $\times$ | $\times$ | $\surd$ | $n$ |
| Ge and Moura [10] | $*$ | $*$ | $\odot$ | $\surd$ | $\times$ | $\surd$ | $n$ |
| Habermehl et al. [14] | $\mathcal{T}_{\mathbb{N}}$ | $\mathcal{T}_{\mathbb{N}}$ | $\odot$ | $\times$ | $\surd$ | $\surd$ | 1 |
| Wintersteiger et al. [28] | bv | bv | $\surd$ | $\surd$ | $\times$ | $\surd$ | 1 |
| This work | $\mathcal{T}_{\mathbb{N}}$ | $*$ | $\surd$ | $\surd$ | $\surd$ | $\surd$ | 1 |

if quantifiers are supported. "$\odot$" means quantifiers could be used with restriction. "$a[a[\cdot]]$" represents if nested reads is allowed, "$|\cdot|$" represents if array length could be used as an attribute in the formula, "ext" represents if the theory is extensional and "dim"represents the dimension of arrays.

WS1S is a simple but expressive logic. An example where integer arrays are broken to bits and expressed as finite sets in WS1S is known in [19]. We employ and further develop this idea in this paper. Furthermore, solving decidability problem by reducing to WS1S is not uncommon. In [23], a decision procedure for bit-vectors by mapping a bit-vector to a set in WS1S is presented.

## 8 Conclusion

In this paper, we investigated a new array theory UABE. It is quite expressive and decidable. A decision procedure is given by translating UABE into WS1S and its soundness and completeness are proven. We also proved that two natural extensions of UABE lead to undecidability. Experimental results show the usage of UABE in program verification.

## References

1. Bofill, M., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: A write-based solver for SAT modulo the theory of arrays. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design, pp. 14:1–14:8. IEEE Press, Piscataway (2008)
2. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Proceedings of the International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 5643, pp. 157–172. Springer Berlin Heidelberg (2009)
3. Bradley, A., Manna, Z., Sipma, H.: What's decidable about arrays? In: Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 3855, pp. 427–442. Springer Berlin Heidelberg (2006)

4. Brummayer, R., Biere, A.: Lemmas on demand for the extensional theory of arrays. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. SMT '08/BPR '08, pp. 6–11. ACM (2008)
5. Brummayer, R., Biere, A.: Boolector: an efficient SMT solver for bit-vectors and arrays. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 5505, pp. 174–177. Springer Berlin Heidelberg (2009)
6. Büchi, J.R.: Weak second-order arithmetic and finite automata. Math. Log. Q. **6**(1–6), 66–92 (1960)
7. de Moura, L., Bjorner, N.: Generalized, efficient array decision procedures. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design, pp. 45–52 (2009)
8. Furia, C.A.: What's decidable about sequences? In: Proceedings of the International Conference on Automated Technology for Verification and Analysis. Lecture Notes in Computer Science, vol. 6252, pp. 128–142. Springer Berlin Heidelberg (2010)
9. Ganesh, V., Dill, D.: A decision procedure for bit-vectors and arrays. In: Proceedings of the International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 4590, pp. 519–531. Springer Berlin Heidelberg (2007)
10. Ge, Y., Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Proceedings of the International Conference on Computer Aided Verification. Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer Berlin Heidelberg (2009)
11. Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. In: Proceedings of the International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 4603, pp. 167–182. Springer Berlin Heidelberg (2007)
12. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Decision procedures for extensions of the theory of arrays. Ann. Math. Artif. Intell. **50**, 231–254 (2007)
13. Goel, A., Krstić, S., Fuchs, A.: Deciding array formulas with frugal axiom instantiation. In: Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning. SMT '08/BPR '08, pp. 12–17. ACM, New York (2008)
14. Habermehl, P., Iosif, R., Vojnar, T.: A. logic of singly indexed arrays. In: Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. Lecture Notes in Computer Science, vol. 5330, pp. 558–573. Springer, Berlin, Heidelberg (2008)
15. Halpern, J.Y. (1991) Presburger arithmetic with unary predicates is $\Pi^1_1$ complete. J. Symb. Log. **56**, 637–642
16. Henriksen, J.G., Jensen, O.J., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.B.: Mona: Monadic second-order logic in practice. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 1019. Springer (1995)
17. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
18. Kapur, D., Zarba, C.: A reduction approach to decision procedures. Tech. rep. (2005)
19. Klarlund, N.: Mona & fido: the logic-automaton connection in practice. In: Conference on Computer Science Logic. Lecture Notes in Computer Science, vol. 1414, pp. 311–326. Springer (1997)
20. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS. Department of Computer Science, Aarhus University, notes Series NS-01-1. Revision of BRICS NS-98-3. Available from http://www.brics.dk/mona/ (2001)
21. Matiyasevich, Y.: Enumerable sets are diophantine. Dokl. Akad. Nauk SSSR **191**(2), 279–282 (1970)
22. McCarthy, J.: Towards a mathematical science of computation. In: IFIP (International Federation for Information Processing), pp. 21–28 . Congress, North-Holland (1962)
23. Möller, M., Rueß, H.: Solving bit-vector equations. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design, pp. 524–524. Springer (1998)
24. Moura, L., Bjrner, N.: Efficient E-Matching for smt solvers. In: Proceedings of International Conference on Automated Deduction. Lecture Notes in Computer Science, vol. 4603, pp. 183–198. Springer Berlin Heidelberg (2007)
25. Nelson, C.G.: Techniques for program verification. PhD. thesis, Stanford University, Stanford (1980)

26. Stump, A., Barrett, C., Dill, D., Levitt, J.: A decision procedure for an extensional theory of arrays. In: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science, pp. 29–37. IEEE Computer Society, Washington (2001)
27. Suzuki, N., Jefferson, D.: Verification decidability of presburger array programs. J. ACM **27**(1), 191–205 (1980)
28. Wintersteiger, C., Hamadi, Y., de Moura, L.: Efficiently solving quantified bit-vector formulas. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design, pp. 239–246 (2010)
29. Zhou, M., He, F., Wang, B., Gu, M.: On array theory of bounded elements. In: Proceedings of International Conference on Computer Aided Verification, pp. 570–584. Springer (2010)