Quantified Array Fragments: Decision Results and Applications

Silvio Ghilardi

Università degli Studi di Milano
Dipartimento di Matematica

Dagstuhl, September 14, 2015

Aim of the talk

This is joint work with **Francesco Alberti** and **Natasha Sharygina** [FroCoS 13, TACAS 14, ATVA 14, FroCoS 15].

The talk will report recent research on *quantified array properties*, including:

- decision/complexity results;
- model-checking applications;
- current status of implementations.

Aim of the talk

This is joint work with **Francesco Alberti** and **Natasha Sharygina** [FroCoS 13, TACAS 14, ATVA 14, FroCoS 15].

The talk will report recent research on *quantified array properties*, including:

- decision/complexity results;
- model-checking applications;
- current status of implementations.

Aim of the talk

This is joint work with **Francesco Alberti** and **Natasha Sharygina** [FroCoS 13, TACAS 14, ATVA 14, FroCoS 15].

The talk will report recent research on *quantified array properties*, including:

- decision/complexity results;
- model-checking applications;
- current status of implementations.

Aim of the talk

This is joint work with **Francesco Alberti** and **Natasha Sharygina** [FroCoS 13, TACAS 14, ATVA 14, FroCoS 15].

The talk will report recent research on *quantified array properties*, including:

- decision/complexity results;
- model-checking applications;
- current status of implementations.

Aim of the talk

This is joint work with **Francesco Alberti** and **Natasha Sharygina** [FroCoS 13, TACAS 14, ATVA 14, FroCoS 15].

The talk will report recent research on *quantified array properties*, including:

- decision/complexity results;
- model-checking applications;
- current status of implementations.

How quantifiers may arise

(an example)

```
        procedure Find( a[L] , e ) {
I_I         i = 0;
I_L         while ( i < L ∧ a[i] ≠ e ) {
              i = i + 1;
            }
I_F         assert ( ∀x.(0 ≤ x < i) → a[x] ≠ e );
        }
```

Can we *decide* safety of this program automatically?

How quantifiers may arise

(an example)

```
        procedure Find( a[L] , e ) {
l_I         i = 0;
l_L         while ( i < L ∧ a[i] ≠ e ) {
              i = i + 1;
            }
l_F         assert ( ∀x.(0 ≤ x < i) → a[x] ≠ e );
        }
```

Can we *decide* safety of this program automatically?

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

Number of iterations

The guard is satisfied for all iterations

Do the "jump"

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

Number of iterations

The guard is satisfied for all iterations

$\Downarrow$

Do the "jump"

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

Number of iterations

The guard is satisfied for all iterations

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \wedge \\ \forall j.(\ \mathtt{i} \leq j < \mathtt{i} + y \ \rightarrow \ j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \wedge \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

Do the "jump"

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

Number of iterations

The guard is satisfied for all iterations

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \wedge \\ \forall j.(\ \mathtt{i} \leq j < \mathtt{i} + y \ \rightarrow \ j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \wedge \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

Do the "jump"

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

Number of iterations

The guard is satisfied for all iterations

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \wedge \\ \forall j.(\ \mathtt{i} \leq j < \mathtt{i} + y \ \rightarrow \ j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \wedge \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

Do the "jump"

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

Number of iterations

The guard is satisfied for all iterations

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \wedge \\ \forall j. (\ \mathtt{i} \leq j < \mathtt{i} + y \ \rightarrow \ j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \wedge \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

Do the "jump"

'Vertical acceleration'

$$\tau_1 := pc = l_L \quad \wedge \quad \underbrace{\mathtt{i} < \mathtt{L} \wedge \mathtt{a}[\mathtt{i}] \neq \mathtt{e}}_{\text{guard}} \quad \wedge \quad \underbrace{\mathtt{i}' = \mathtt{i} + 1}_{\text{update}}$$

$$\Downarrow$$

Number of iterations

The guard is satisfied for all iterations

$$\tau_1^+ := \exists y. \begin{pmatrix} y > 0 \wedge pc = l_L \wedge \\ \forall j. (\ \mathtt{i} \leq j < \mathtt{i} + y \ \rightarrow \ j < \mathtt{L} \wedge \mathtt{a}[j] \neq \mathtt{e}\ ) \wedge \\ \mathtt{i}' = \mathtt{i} + y \end{pmatrix}$$

Do the "jump"

How quantifiers may arise

(another example)

$$
\begin{array}{ll}
 & \text{procedure Max}(\ \mathtt{a}[\mathtt{L}+1])\ \{ \\
l_I & \quad \text{int}\ \ M[\mathtt{L}+1]; \qquad\qquad \text{int}\ \ \mathtt{i}; \\
 & \quad M[0] = a[0]; \qquad \mathtt{i} = 0; \\
l_L & \quad \text{while}\ (\ \mathtt{i} < \mathtt{L})\ \{ \\
 & \qquad M[\mathtt{i}+1] = max(a[\mathtt{i}+1], M[\mathtt{i}]); \\
 & \qquad \mathtt{i} = \mathtt{i} + 1; \\
 & \quad \} \\
l_F & \quad \text{assert}\ (\ \forall x.(0 \leq x \leq \mathtt{L}) \rightarrow \mathtt{a}[x] \leq M[\mathtt{L}]\ ); \\
 & \}
\end{array}
$$

'Vertical acceleration'

The acceleration of the loop gives the following formula

$$\tau_1^+ := \exists y. \left(\begin{array}{l} y > 0 \wedge pc = l_L \wedge \\ \wedge\ \forall j.(\ \mathtt{i} \leq j < \mathtt{i} + y \ \rightarrow\ j < \mathtt{L} \wedge \\ \quad \wedge\ \mathtt{M}'[j+1] = max(\mathtt{a}[j+1], \mathtt{M}[j]) \wedge \\ \wedge\ \forall j.(\ \mathtt{i} > j \vee j \geq \mathtt{i} + y \ \rightarrow\ \mathtt{M}'[j+1] = \mathtt{M}[j+1]) \\ \wedge\ \mathtt{a}' = \mathtt{a} \wedge \mathtt{i}' = \mathtt{i} + y \end{array} \right)$$

Notice that, since the values of M change, we had to be more careful here and to write τ_1^+ as a formula relating the values of

$$\mathtt{M},\ \mathtt{M}',\ \mathtt{a},\ \mathtt{a}',\ \mathtt{i},\ \mathtt{i}'$$

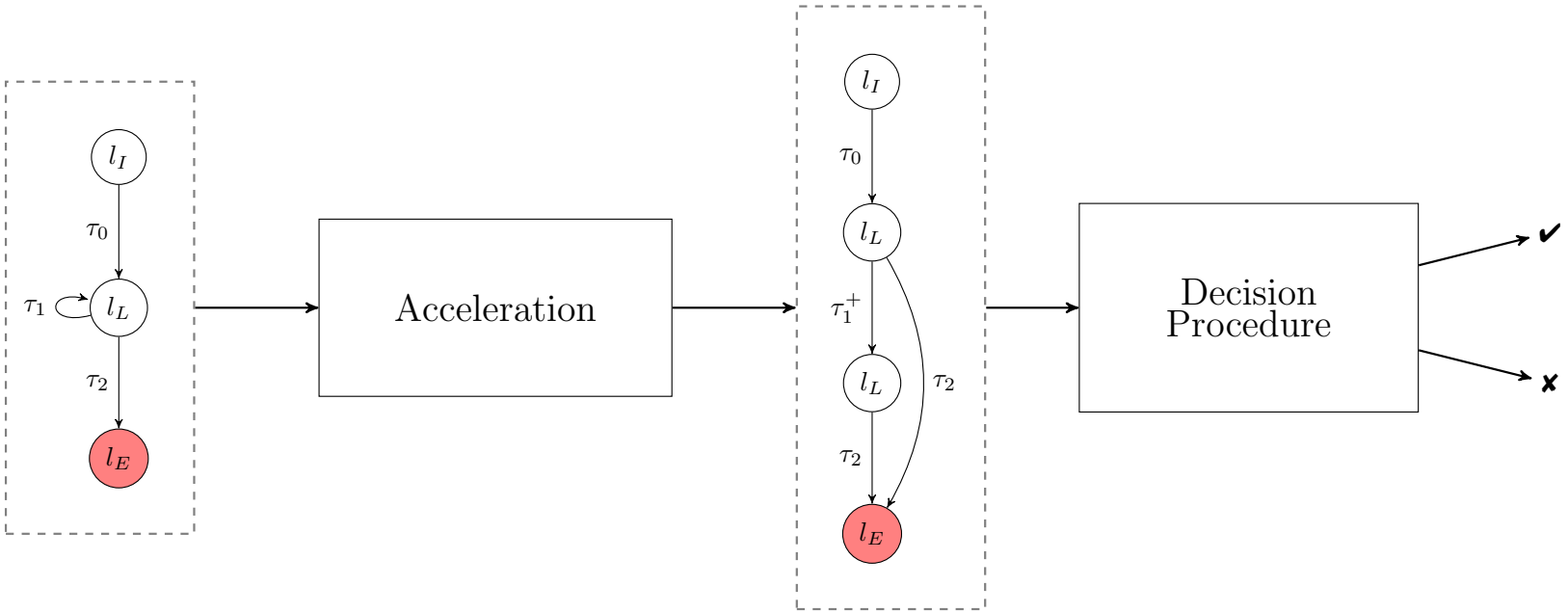
(i.e. the values of $\mathtt{M}, \mathtt{a}, \mathtt{i}$ before and after a finite number of executions of the loop).

How quantifiers may arise

Problem:

- Infinitely many paths to analyze because of loops bounded by symbolic constants (e.g., L, the length of the array)

Idea:



Outline

Outline

1. Decidable Fragments
   - The Bradley-Manna-Sipma fragment
   - The flat mono-sorted fragment
   - The flat two-sorted fragment
   - The SIL fragment
   - Our Contribution to FroCoS 2015
2. Applications and Implementations
   - Array Acceleration
   - MCMT and Booster

1. Decidable Fragments
   - The Bradley-Manna-Sipma fragment
   - The flat mono-sorted fragment
   - The flat two-sorted fragment
   - The SIL fragment
   - Our Contribution to FroCoS 2015

2. Applications and Implementations
   - Array Acceleration
   - MCMT and Booster

Notation

We work in a decidable fragment of arithmetic (typically a definable extension of Presburger arithmetic); we expand the related language with *free constants* and *free function symbols*. When we speak about truth or about validity, *we refer to the structures having as reduct the standard model of the integers* with the natural interpretation of the arithmetic symbols.

A *(purely) arithmetical* formula/term is a formula/term that does not contain free function symbols (notice however that such a formula/term may contain *parameters*, i.e. free constants).

Notation $\alpha(\underline{x})$ means that the free variables of α are included in the tuple $\underline{x}$ (again, α may contain also parameters).

Notation

We work in a decidable fragment of arithmetic (typically a definable extension of Presburger arithmetic); we expand the related language with *free constants* and *free function symbols*. When we speak about truth or about validity, *we refer to the structures having as reduct the standard model of the integers* with the natural interpretation of the arithmetic symbols.

A *(purely) arithmetical* formula/term is a formula/term that does not contain free function symbols (notice however that such a formula/term may contain *parameters*, i.e. free constants).

Notation $\alpha(\underline{x})$ means that the free variables of α are included in the tuple $\underline{x}$ (again, α may contain also parameters).

Notation

We work in a decidable fragment of arithmetic (typically a definable extension of Presburger arithmetic); we expand the related language with *free constants* and *free function symbols*. When we speak about truth or about validity, *we refer to the structures having as reduct the standard model of the integers* with the natural interpretation of the arithmetic symbols.

A *(purely) arithmetical* formula/term is a formula/term that does not contain free function symbols (notice however that such a formula/term may contain *parameters*, i.e. free constants).

Notation $\alpha(\underline{x})$ means that the free variables of α are included in the tuple $\underline{x}$ (again, α may contain also parameters).

The Problem

We may use underlined or bold notation for tuples of terms. Free function symbols denote arrays, with read operation (function application) written $a[t]$.

Arrays are equipped with length; for simplicity, we assume that length is the same for all arrays we consider; we represent it by a free constant N. We are not interested on the value $a[x]$ for any x outside the interval $[0, N]$, when discussing satisfiability of our formulae (thus we assume, say, $a[x] = 0$ for $x \notin [0, N]$.

Adding free unary function symbols to Presburger arithmetics makes it highly undecidable; quantifier-free formulae are on the other hand decidable by Nelson-Oppen combination.

The question is: how far can we go beyond quantifier-free case?

The Problem

We may use underlined or bold notation for tuples of terms. Free function symbols denote arrays, with read operation (function application) written $a[t]$.

Arrays are equipped with length; for simplicity, we assume that length is the same for all arrays we consider; we represent it by a free constant N. We are not interested on the value $a[x]$ for any x outside the interval $[0, N]$, when discussing satisfiability of our formulae (thus we assume, say, $a[x] = 0$ for $x \notin [0, N]$.

Adding free unary function symbols to Presburger arithmetics makes it highly undecidable; quantifier-free formulae are on the other hand decidable by Nelson-Oppen combination.

The question is: how far can we go beyond quantifier-free case?

The Problem

We may use underlined or bold notation for tuples of terms. Free function symbols denote arrays, with read operation (function application) written $a[t]$.

Arrays are equipped with length; for simplicity, we assume that length is the same for all arrays we consider; we represent it by a free constant N. We are not interested on the value $a[x]$ for any x outside the interval $[0, N]$, when discussing satisfiability of our formulae (thus we assume, say, $a[x] = 0$ for $x \notin [0, N]$.

Adding free unary function symbols to Presburger arithmetics makes it highly undecidable; quantifier-free formulae are on the other hand decidable by Nelson-Oppen combination.

The question is: how far can we go beyond quantifier-free case?

The Problem

We may use underlined or bold notation for tuples of terms. Free function symbols denote arrays, with read operation (function application) written $a[t]$.

Arrays are equipped with length; for simplicity, we assume that length is the same for all arrays we consider; we represent it by a free constant N. We are not interested on the value $a[x]$ for any x outside the interval $[0, N]$, when discussing satisfiability of our formulae (thus we assume, say, $a[x] = 0$ for $x \notin [0, N]$.

Adding free unary function symbols to Presburger arithmetics makes it highly undecidable; quantifier-free formulae are on the other hand decidable by Nelson-Oppen combination.

The question is: how far can we go beyond quantifier-free case?

BMS-fragment

Definition

A formula is in the *array property fragment* [BMS 06] iff it is equivalent to a disjunction of formulae of the kind $\psi_1 \wedge \cdots \wedge \psi_n$, where each ψ_i is either a literal or a BMS-guard.

A BMS-guard is a formula of the kind $\forall \underline{j}\ (G \rightarrow \theta)$, where

- G is a conjunction of atoms of the kind

$$j_1 \leq j_2,\ j_1 \leq t,\ t \leq j_1$$

for $j_1, j_2 \in \underline{j}$ and t arithmetical;

- θ is obtained from a quantifier-free arithmetical formula $\alpha(\underline{x})$ by replacing the variables $\underline{x}$ by terms of the kind $a[\underline{t}]$, $a[\underline{j'}]$ (for $\underline{j'} \subseteq \underline{j}$ and $\underline{t}$ arithmetical).

BMS-fragment

Definition

A formula is in the *array property fragment* [BMS 06] iff it is equivalent to a disjunction of formulae of the kind $\psi_1 \wedge \cdots \wedge \psi_n$, where each ψ_i is either a literal or a BMS-guard.

A BMS-guard is a formula of the kind $\forall \underline{j}\ (G \rightarrow \theta)$, where

- G is a conjunction of atoms of the kind

$$j_1 \leq j_2,\ j_1 \leq t,\ t \leq j_1$$

  for $j_1, j_2 \in \underline{j}$ and t arithmetical;
- θ is obtained from a quantifier-free arithmetical formula $\alpha(\underline{x})$ by replacing the variables $\underline{x}$ by terms of the kind $a[\underline{t}], a[\underline{j'}]$ (for $\underline{j'} \subseteq \underline{j}$ and $\underline{t}$ arithmetical).

BMS-fragment

Definition

A formula is in the *array property fragment* [BMS 06] iff it is equivalent to a disjunction of formulae of the kind $\psi_1 \wedge \cdots \wedge \psi_n$, where each ψ_i is either a literal or a BMS-guard.

A BMS-guard is a formula of the kind $\forall \underline{j}\ (G \rightarrow \theta)$, where

- G is a conjunction of atoms of the kind

$$j_1 \leq j_2,\ j_1 \leq t,\ t \leq j_1$$

  for $j_1, j_2 \in \underline{j}$ and t arithmetical;
- θ is obtained from a quantifier-free arithmetical formula $\alpha(\underline{x})$ by replacing the variables $\underline{x}$ by terms of the kind $a[\underline{t}], a[\underline{j}']$ (for $\underline{j}' \subseteq \underline{j}$ and $\underline{t}$ arithmetical).

BMS-fragment: features

- Many universally quantified variables in guards allowed;
- many-dimensional arrays (i.e. n-free function symbols) allowed too;
- universally quantified variables can occur in guard consequents only inside reads;
- satisfiability decision procedure basically via instantiation: an interval partition of relevant arithmetic terms is guessed and arrays which are constants on such intervals are produced;
- complexity NExpTime (but it reduces to NP if one fixes the number of universally quantified variables).

BMS-fragment: features

- Many universally quantified variables in guards allowed;
- many-dimensional arrays (i.e. n-free function symbols) allowed too;
- universally quantified variables can occur in guard consequents only inside reads;
- satisfiability decision procedure basically via instantiation: an interval partition of relevant arithmetic terms is guessed and arrays which are constants on such intervals are produced;
- complexity NExpTime (but it reduces to NP if one fixes the number of universally quantified variables).

BMS-fragment: features

- Many universally quantified variables in guards allowed;
- many-dimensional arrays (i.e. n-free function symbols) allowed too;
- universally quantified variables can occur in guard consequents only inside reads;
- satisfiability decision procedure basically via instantiation: an interval partition of relevant arithmetic terms is guessed and arrays which are constants on such intervals are produced;
- complexity NExpTime (but it reduces to NP if one fixes the number of universally quantified variables).

BMS-fragment: features

- Many universally quantified variables in guards allowed;
- many-dimensional arrays (i.e. n-free function symbols) allowed too;
- universally quantified variables can occur in guard consequents only inside reads;
- satisfiability decision procedure basically via instantiation: an interval partition of relevant arithmetic terms is guessed and arrays which are constants on such intervals are produced;
- complexity NExpTime (but it reduces to NP if one fixes the number of universally quantified variables).

BMS-fragment: features

- Many universally quantified variables in guards allowed;
- many-dimensional arrays (i.e. *n*-free function symbols) allowed too;
- universally quantified variables can occur in guard consequents only inside reads;
- satisfiability decision procedure basically via instantiation: an interval partition of relevant arithmetic terms is guessed and arrays which are constants on such intervals are produced;
- complexity NExpTime (but it reduces to NP if one fixes the number of universally quantified variables).

The flat1 fragment

A formula or a term are said to be *flat* iff for every term of the kind $a[t]$ occurring in them, the sub-term t is always a variable or a parameter. Notice that every formula is logically equivalent to a flat one; however the flattening transformations may alter the quantifiers prefix of a formula in prenex form.

Theorem

Satisfiability of $\exists^*\forall$-flat sentences is decidable whenever satisfiability of $\exists^*\forall\exists^*$ purely arithmetic sentences is decidable.

The above result is modular: it applies to Presburger arithmetics, but also to stronger decidable fragments of arithmetics (like linear arithmetics with exponentiation).

The flat1 fragment

A formula or a term are said to be *flat* iff for every term of the kind $a[t]$ occurring in them, the sub-term t is always a variable or a parameter. Notice that every formula is logically equivalent to a flat one; however the flattening transformations may alter the quantifiers prefix of a formula in prenex form.

Theorem

Satisfiability of $\exists^*\forall$-flat sentences is decidable whenever satisfiability of $\exists^*\forall\exists^*$ purely arithmetic sentences is decidable.

The above result is modular: it applies to Presburger arithmetics, but also to stronger decidable fragments of arithmetics (like linear arithmetics with exponentiation).

The flat1 fragment

A formula or a term are said to be *flat* iff for every term of the kind $a[t]$ occurring in them, the sub-term t is always a variable or a parameter. Notice that every formula is logically equivalent to a flat one; however the flattening transformations may alter the quantifiers prefix of a formula in prenex form.

Theorem

Satisfiability of $\exists^*\forall$-flat sentences is decidable whenever satisfiability of $\exists^*\forall\exists^*$ purely arithmetic sentences is decidable.

The above result is modular: it applies to Presburger arithmetics, but also to stronger decidable fragments of arithmetics (like linear arithmetics with exponentiation).

The flat1 fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by reverse skolemization: a quadratic instance of a Σ_2^0-purely arithmetical problem is produced;
- ... thus instantiation techniques are not sufficient (quantifier-elimination used instead?).

The flat1 fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by reverse skolemization: a quadratic instance of a Σ^0_2-purely arithmetical problem is produced;
- ... thus instantiation techniques are not sufficient (quantifier-elimination used instead?).

The flat1 fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by reverse skolemization: a quadratic instance of a Σ_2^0-purely arithmetical problem is produced;
- ... thus instantiation techniques are not sufficient (quantifier-elimination used instead?).

The flat1 fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by reverse skolemization: a quadratic instance of a Σ_2^0-purely arithmetical problem is produced;
- ... thus instantiation techniques are not sufficient (quantifier-elimination used instead?).

The monic-flat fragment

We may view array theory as a two-sorted theory (we have one sort for indexes and one sort for elements/data).

Both sorts are constrained by a fragment of arithmetics; the fragment can be the same, but in the two-sorted approach some expressive power is lost: an atom like $a[i] < i$ is not allowed anymore, it cannot be written. Depending on the sort of the root predicate, atoms can be classified as *index* or *elem* atoms.

Applications suggest to restrict quantified variables (but not parameters) to variables for indexes (we shall follow this suggestion).

The monic-flat fragment

We may view array theory as a two-sorted theory (we have one sort for indexes and one sort for elements/data).

Both sorts are constrained by a fragment of arithmetics; the fragment can be the same, but in the two-sorted approach some expressive power is lost: an atom like $a[i] < i$ is not allowed anymore, it cannot be written. Depending on the sort of the root predicate, atoms can be classified as *index* or *elem* atoms.

Applications suggest to restrict quantified variables (but not parameters) to variables for indexes (we shall follow this suggestion).

The monic-flat fragment

We may view array theory as a two-sorted theory (we have one sort for indexes and one sort for elements/data).

Both sorts are constrained by a fragment of arithmetics; the fragment can be the same, but in the two-sorted approach some expressive power is lost: an atom like $a[i] < i$ is not allowed anymore, it cannot be written. Depending on the sort of the root predicate, atoms can be classified as *index* or *elem* atoms.

Applications suggest to restrict quantified variables (but not parameters) to variables for indexes (we shall follow this suggestion).

The monic-flat fragment

A sentence in the language of is said to be *monic* iff it is in prenex form and every index atom occurring in it contains at most one variable falling within the scope of a universal quantifier.

$$(I)\ \forall i_1 \forall i_2 . (i_1 \leq i_2 \rightarrow a[i_1] \leq a[i_2]); \quad (\text{flat, not monic})$$
$$(II)\ c_1 \leq c_2 \wedge a[c_1] \not\leq a[c_2]; \quad (\text{flat, monic})$$
$$(III)\ \forall i_1 \forall i_2 . a[i_1] = a[i_2]; \quad (\text{flat, monic})$$
$$(IV)\ \forall i . (i \equiv_2 0 \rightarrow a[i] = 0) \quad (\text{flat, monic})$$

Theorem

If satisfiability of $\exists^*\forall$-purely arithmetic sentences is decidable, then satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable.

The monic-flat fragment

A sentence in the language of is said to be *monic* iff it is in prenex form and every index atom occurring in it contains at most one variable falling within the scope of a universal quantifier.

$$
\begin{aligned}
&(I)\ \forall i_1 \forall i_2 . (i_1 \leq i_2 \rightarrow a[i_1] \leq a[i_2]); && (\text{flat, not monic})\\
&(II)\ c_1 \leq c_2 \wedge a[c_1] \not\leq a[c_2]; && (\text{flat, monic})\\
&(III)\ \forall i_1 \forall i_2 . a[i_1] = a[i_2]; && (\text{flat, monic})\\
&(IV)\ \forall i . (i \equiv_2 0 \rightarrow a[i] = 0) && (\text{flat, monic})
\end{aligned}
$$

Theorem

If satisfiability of $\exists^*\forall$-purely arithmetic sentences is decidable, then satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable.

The monic-flat fragment

A sentence in the language of is said to be *monic* iff it is in prenex form and every index atom occurring in it contains at most one variable falling within the scope of a universal quantifier.

$$
\begin{aligned}
&(I)\ \forall i_1 \forall i_2 . (i_1 \leq i_2 \rightarrow a[i_1] \leq a[i_2]); && (\text{flat, not monic})\\
&(II)\ c_1 \leq c_2 \wedge a[c_1] \not\leq a[c_2]; && (\text{flat, monic})\\
&(III)\ \forall i_1 \forall i_2 . a[i_1] = a[i_2]; && (\text{flat, monic})\\
&(IV)\ \forall i . (i \equiv_2 0 \rightarrow a[i] = 0) && (\text{flat, monic})
\end{aligned}
$$

Theorem

If satisfiability of $\exists^*\forall$-purely arithmetic sentences is decidable, then satisfiability of $\exists^*\forall^*$-monic-flat sentences is decidable.

The monic flat-fragment: features

- Allows simultaneous controlled reference to many universally quantified variables at a time (still disallow many-dimensional arrays);
- by definition, no simultaneous reference (in the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by guessing index types (i.e. maximal consistent sets of index atoms) that are realized in the model;
- this makes satisfiability NExpTime-complete (by reduction to exponentially bounded domino systems).

The monic flat-fragment: features

- Allows simultaneous controlled reference to many universally quantified variables at a time (still disallow many-dimensional arrays);
- by definition, no simultaneous reference (in the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by guessing index types (i.e. maximal consistent sets of index atoms) that are realized in the model;
- this makes satisfiability NExpTime-complete (by reduction to exponentially bounded domino systems).

The monic flat-fragment: features

- Allows simultaneous controlled reference to many universally quantified variables at a time (still disallow many-dimensional arrays);
- by definition, no simultaneous reference (in the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by guessing index types (i.e. maximal consistent sets of index atoms) that are realized in the model;
- this makes satisfiability NExpTime-complete (by reduction to exponentially bounded domino systems).

The monic flat-fragment: features

- Allows simultaneous controlled reference to many universally quantified variables at a time (still disallow many-dimensional arrays);
- by definition, no simultaneous reference (in the same literal) to a universally quantified variable j and to its content $a[j]$ is allowed;
- satisfiability decision procedure is by guessing index types (i.e. maximal consistent sets of index atoms) that are realized in the model;
- this makes satisfiability NExpTime-complete (by reduction to exponentially bounded domino systems).

SIL-fragment

In [Habermehl, Iosif, Vojnar LPAR 08] (see also other papers by the same group of authors) another interesting decidable fragment is introduced.

The peculiarity of this fragment (SIL, for 'single index logic') is that of allowing both sub-terms of the kind $a[i]$ and of the kind $a[i+1]$, where i is the (unique) universally quantified variable that can be used. This liberality is compensated by other restrictions, on the arithmetic expressions and on the use of disjunctions.

The syntax of the so-called SIL-fragment is rather elaborated, however there is a simpler normalized equivalent formulation the authors themselves use after a preprocessing step (Lemma 6, in Section 5.1 of above paper).

SIL-fragment

In [Habermehl, Iosif, Vojnar LPAR 08] (see also other papers by the same group of authors) another interesting decidable fragment is introduced.

The peculiarity of this fragment (SIL, for 'single index logic') is that of allowing both sub-terms of the kind $a[i]$ and of the kind $a[i+1]$, where i is the (unique) universally quantified variable that can be used. This liberality is compensated by other restrictions, on the arithmetic expressions and on the use of disjunctions.

The syntax of the so-called SIL-fragment is rather elaborated, however there is a simpler normalized equivalent formulation the authors themselves use after a preprocessing step (Lemma 6, in Section 5.1 of above paper).

SIL-fragment

In [Habermehl, Iosif, Vojnar LPAR 08] (see also other papers by the same group of authors) another interesting decidable fragment is introduced.

The peculiarity of this fragment (SIL, for 'single index logic') is that of allowing both sub-terms of the kind $a[i]$ and of the kind $a[i+1]$, where i is the (unique) universally quantified variable that can be used. This liberality is compensated by other restrictions, on the arithmetic expressions and on the use of disjunctions.

The syntax of the so-called SIL-fragment is rather elaborated, however there is a simpler normalized equivalent formulation the authors themselves use after a preprocessing step (Lemma 6, in Section 5.1 of above paper).

SIL-fragment

The normalized formulation covers Boolean combinations of ground formulae and of guards of the following two types

$$\forall i\,(t \leq i < u \rightarrow \nu) \tag{1}$$

$$\forall i\,(t \leq i < u) \wedge i \equiv_n m \rightarrow \nu) \tag{2}$$

where: (i) t, u are ground arithmetical terms; (ii) $m, n \in \mathbb{N}$; (iii) ν is a *conjunction* of literals of the kinds

$$a_1[i] \sim \ell, \quad a_1[i] - i \sim n, \quad a_1[i] - a_2[i+1] \sim n$$

for $\sim \in \{\leq, \geq\}$, ℓ ground arithmetical, $n \in \mathbb{Z}$.

SIL-fragment

The normalized formulation covers Boolean combinations of ground formulae and of guards of the following two types

$$\forall i\,(t \leq i < u \rightarrow \nu) \tag{1}$$

$$\forall i\,(t \leq i < u) \wedge i \equiv_n m \rightarrow \nu) \tag{2}$$

where: (i) t, u are ground arithmetical terms; (ii) $m, n \in \mathbb{N}$; (iii) ν is a *conjunction* of literals of the kinds

$$a_1[i] \sim \ell, \quad a_1[i] - i \sim n, \quad a_1[i] - a_2[i+1] \sim n$$

for $\sim \in \{\leq, \geq\}$, ℓ ground arithmetical, $n \in \mathbb{Z}$.

SIL-fragment

The normalized formulation covers Boolean combinations of ground formulae and of guards of the following two types

$$\forall i\ (t \leq i < u \rightarrow \nu) \tag{1}$$
$$\forall i\ (t \leq i < u) \wedge i \equiv_n m \rightarrow \nu) \tag{2}$$

where: (i) t, u are ground arithmetical terms; (ii) $m, n \in \mathbb{N}$; (iii) ν is a *conjunction* of literals of the kinds

$$a_1[i] \sim \ell, \quad a_1[i] - i \sim n, \quad a_1[i] - a_2[i+1] \sim n$$

for $\sim \in \{\leq, \geq\}$, ℓ ground arithmetical, $n \in \mathbb{Z}$.

SIL-fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to terms j, $a[j]$, $a[j+1]$ is allowed;
- atoms in consequents of guards have restricted difference logic-shape form;
- disjunction is not allowed in such consequents;
- satisfiability decision procedure is by back-and-forth conversion between logic and automata,
- ... thus not very suitable for integration into a declarative SMT-based context.

SIL-fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to terms j, $a[j]$, $a[j+1]$ is allowed;
- atoms in consequents of guards have restricted difference logic-shape form;
- disjunction is not allowed in such consequents;
- satisfiability decision procedure is by back-and-forth conversion between logic and automata,
- ... thus not very suitable for integration into a declarative SMT-based context.

SIL-fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to terms j, $a[j]$, $a[j+1]$ is allowed;
- atoms in consequents of guards have restricted difference logic-shape form;
- disjunction is not allowed in such consequents;
- satisfiability decision procedure is by back-and-forth conversion between logic and automata,
- ... thus not very suitable for integration into a declarative SMT-based context.

SIL-fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to terms j, $a[j]$, $a[j+1]$ is allowed;
- atoms in consequents of guards have restricted difference logic-shape form;
- disjunction is not allowed in such consequents;
- satisfiability decision procedure is by back-and-forth conversion between logic and automata,
- ... thus not very suitable for integration into a declarative SMT-based context.

SIL-fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to terms j, $a[j]$, $a[j+1]$ is allowed;
- atoms in consequents of guards have restricted difference logic-shape form;
- disjunction is not allowed in such consequents;
- satisfiability decision procedure is by back-and-forth conversion between logic and automata,
- ... thus not very suitable for integration into a declarative SMT-based context.

SIL-fragment: features

- Does not allow more than one universally quantified variable nor many-dimensional arrays;
- simultaneous reference (within the same literal) to terms j, $a[j]$, $a[j+1]$ is allowed;
- atoms in consequents of guards have restricted difference logic-shape form;
- disjunction is not allowed in such consequents;
- satisfiability decision procedure is by back-and-forth conversion between logic and automata,
- ... thus not very suitable for integration into a declarative SMT-based context.

Revisiting SIL-fragment

The above analysis of the SIL-fragment relies on acceleratability results for difference bound constraints.

The idea is to reinterpret SIL-decision procedure in a modular and declarative way, making it more general and appropriate to SMT approaches. Doing this, we were able to cover by our new results [FroCoS 2015] also the flat mono-sorted fragment above.

We first need an abstract axiomatic characterization of an acceleratability result which is suitable for our modular purposes.

Remark By an 'acceleratability result' we mean a definability result for the transitive closure of a class of relations. The acceleratability results we refer here are for pure arithmetic (let us call this 'horizontal acceleration' to distinguish it from 'vertical' acceleration for arrays we saw before).

Revisiting SIL-fragment

The above analysis of the SIL-fragment relies on acceleratability results for difference bound constraints.

The idea is to reinterpret SIL-decision procedure in a modular and declarative way, making it more general and appropriate to SMT approaches. Doing this, we were able to cover by our new results [FroCoS 2015] also the flat mono-sorted fragment above.

We first need an abstract axiomatic characterization of an acceleratability result which is suitable for our modular purposes.

Remark By an 'acceleratability result' we mean a definability result for the transitive closure of a class of relations. The acceleratability results we refer here are for pure arithmetic (let us call this 'horizontal acceleration' to distinguish it from 'vertical' acceleration for arrays we saw before).

Revisiting SIL-fragment

The above analysis of the SIL-fragment relies on acceleratability results for difference bound constraints.

The idea is to reinterpret SIL-decision procedure in a modular and declarative way, making it more general and appropriate to SMT approaches. Doing this, we were able to cover by our new results [FroCoS 2015] also the flat mono-sorted fragment above.

We first need an abstract axiomatic characterization of an acceleratability result which is suitable for our modular purposes.

Remark By an 'acceleratability result' we mean a definability result for the transitive closure of a class of relations. The acceleratability results we refer here are for pure arithmetic (let us call this 'horizontal acceleration' to distinguish it from 'vertical' acceleration for arrays we saw before).

Revisiting SIL-fragment

The above analysis of the SIL-fragment relies on acceleratability results for difference bound constraints.

The idea is to reinterpret SIL-decision procedure in a modular and declarative way, making it more general and appropriate to SMT approaches. Doing this, we were able to cover by our new results [FroCoS 2015] also the flat mono-sorted fragment above.

We first need an abstract axiomatic characterization of an acceleratability result which is suitable for our modular purposes.

Remark By an 'acceleratability result' we mean a definability result for the transitive closure of a class of relations. The acceleratability results we refer here are for pure arithmetic (let us call this 'horizontal acceleration' to distinguish it from 'vertical' acceleration for arrays we saw before).

Acceleratable Fragments

Definition

A purely arithmetical formula $\phi(\underline{x}, \underline{x}')$, is said to be *acceleratable* iff there exists (and one can actually compute) a formula $\phi^*(\underline{x}, \underline{x}', j)$ such that for all $n \in \mathbb{N}$

$$\models \phi^n(\underline{x}, \underline{x}') \leftrightarrow \phi^*(\underline{x}, \underline{x}', n) \tag{3}$$

Definition

A set of purely arithmetical formulae Γ is said to be an *acceleratable fragment* iff every $\phi \in \Gamma$ is acceleratable and Γ is closed under conjunctions and renaming substitutions. An acceleratable fragment Γ is said to be *normal* iff it contains the formulae $x' = x + 1$ and $y' = x'$ for every variables x, y.

Acceleratable Fragments

Definition

A purely arithmetical formula $\phi(\underline{x}, \underline{x}')$, is said to be *acceleratable* iff there exists (and one can actually compute) a formula $\phi^*(\underline{x}, \underline{x}', j)$ such that for all $n \in \mathbb{N}$

$$\models \phi^n(\underline{x}, \underline{x}') \leftrightarrow \phi^*(\underline{x}, \underline{x}', n) \tag{3}$$

Definition

A set of purely arithmetical formulae Γ is said to be an *acceleratable fragment* iff every $\phi \in \Gamma$ is acceleratable and Γ is closed under conjunctions and renaming substitutions. An acceleratable fragment Γ is said to be *normal* iff it contains the formulae $x' = x + 1$ and $y' = x'$ for every variables x, y.

Acceleratable Fragments

Examples of acceleratable fragments include:

- difference bounds constraints (conjunctions of atoms in difference logic);
- octagons (conjunctions of unit-two-variables-per-inequality atoms);
- finite monoid affine transformations;
- iteratable formulae (roughly: relations where primed variables are either randomly updated or updated according to a function whose iterations are parametrically definable - see the precise definition in our paper).

Acceleratable Fragments

Examples of acceleratable fragments include:

- difference bounds constraints (conjunctions of atoms in difference logic);
- octagons (conjunctions of unit-two-variables-per-inequality atoms);
- finite monoid affine transformations;
- iteratable formulae (roughly: relations where primed variables are either randomly updated or updated according to a function whose iterations are parametrically definable - see the precise definition in our paper).

Our Contribution

Let Γ be an acceleratable fragment; a Γ-*guard* is a formula of the kind

$$\forall i \, (\, t \leq i < u \rightarrow \phi(i, \mathbf{a}[i], \mathbf{a}[i+1]) \,) \tag{4}$$

such that the formula $i' = i + 1 \wedge \phi(i, \underline{y}, \underline{y'})$ belongs to Γ and t, u are ground terms.

Theorem

Let Γ be an acceleratable fragment; then, any Boolean combination of ground formulae and Γ-guards is decidable for satisfiability.

Our Contribution

Let Γ be an acceleratable fragment; a Γ-*guard* is a formula of the kind

$$\forall i \, (\, t \leq i < u \rightarrow \phi(i, \mathbf{a}[i], \mathbf{a}[i+1]) \,) \tag{4}$$

such that the formula $i' = i + 1 \wedge \phi(i, \underline{y}, \underline{y'})$ belongs to Γ and t, u are ground terms.

Theorem

Let Γ be an acceleratable fragment; then, any Boolean combination of ground formulae and Γ-guards is decidable for satisfiability.

Our contribution: features

- (under appropriate choice for Γ) includes flat-monosorted and SIL-fragments;
- satisfiability decision procedure guesses an interval partition (like in BMS case), then solves satisfiability in an interval $[t, u)$ using the accelerations ϕ^* with parameter instantiated to $t - u$;
- complexity (after the nondeterministic polynomial step producing the interval partition) is $g \circ f$, where f is the complexity of the algorithm producing ϕ^* and g is the complexity of the arithmetic solver.

Our contribution: features

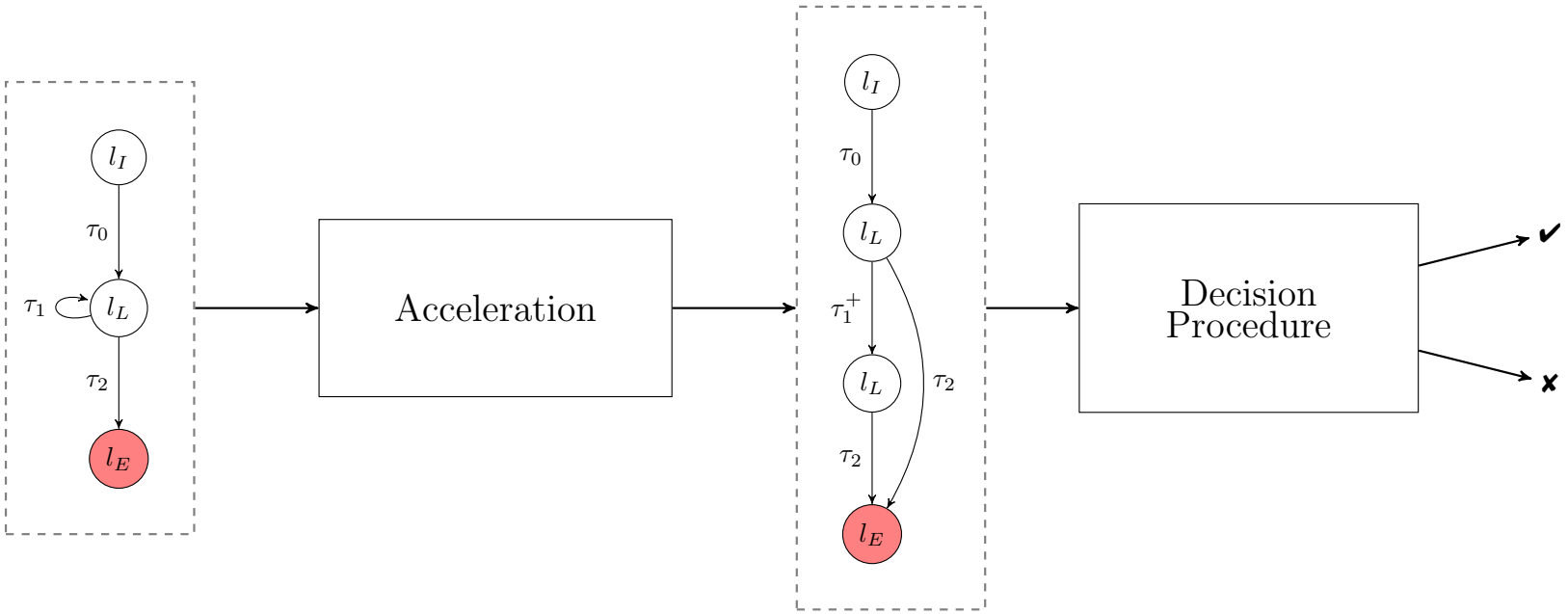
- (under appropriate choice for Γ) includes flat-monosorted and SIL-fragments;
- satisfiability decision procedure guesses an interval partition (like in BMS case), then solves satisfiability in an interval $[t, u)$ using the accelerations ϕ^* with parameter instantiated to $t - u$;
- complexity (after the nondeterministic polynomial step producing the interval partition) is $g \circ f$, where f is the complexity of the algorithm producing ϕ^* and g is the complexity of the arithmetic solver.

Our contribution: features

- (under appropriate choice for Γ) includes flat-monosorted and SIL-fragments;
- satisfiability decision procedure guesses an interval partition (like in BMS case), then solves satisfiability in an interval $[t, u)$ using the accelerations ϕ^* with parameter instantiated to $t - u$;
- complexity (after the nondeterministic polynomial step producing the interval partition) is $g \circ f$, where f is the complexity of the algorithm producing ϕ^* and g is the complexity of the arithmetic solver.

Acceleration for arrays

Let's go back to our picture:



Acceleration for arrays

- for many customary guarded assignments occurring in array programs, τ_1^+ is described by formulae belonging to the above considered quantified array fragments;
- acceleration for arrays can be computed automatically from some relevant source code, see our papers for details;
- whenever array updates refers to the values of contiguous cells (i.e. to $\mathtt{a}[i]$ and $\mathtt{a}[i+1]$ - like in the above Max procedure), BSM and flat fragments are inappropriate: you need the fragment of the present contribution (SIL is often - but not always - sufficient as well);
- the peculiar feature of our proposal is modularity: building a library of (even very small or ad hoc) acceleratable fragments, one increases covered applications.

Acceleration for arrays

- for many customary guarded assignments occurring in array programs, τ_1^+ is described by formulae belonging to the above considered quantified array fragments;
- acceleration for arrays can be computed automatically from some relevant source code, see our papers for details;
- whenever array updates refers to the values of contiguous cells (i.e. to $a[i]$ and $a[i+1]$ - like in the above Max procedure), BSM and flat fragments are inappropriate: you need the fragment of the present contribution (SIL is often - but not always - sufficient as well);
- the peculiar feature of our proposal is modularity: building a library of (even very small or ad hoc) acceleratable fragments, one increases covered applications.

Acceleration for arrays

- for many customary guarded assignments occurring in array programs, τ_1^+ is described by formulae belonging to the above considered quantified array fragments;
- acceleration for arrays can be computed automatically from some relevant source code, see our papers for details;
- whenever array updates refers to the values of contiguous cells (i.e. to $\mathtt{a}[i]$ and $\mathtt{a}[i+1]$ - like in the above Max procedure), BSM and flat fragments are inappropriate: you need the fragment of the present contribution (SIL is often - but not always - sufficient as well);
- the peculiar feature of our proposal is modularity: building a library of (even very small or ad hoc) acceleratable fragments, one increases covered applications.

Acceleration for arrays

- for many customary guarded assignments occurring in array programs, τ_1^+ is described by formulae belonging to the above considered quantified array fragments;
- acceleration for arrays can be computed automatically from some relevant source code, see our papers for details;
- whenever array updates refers to the values of contiguous cells (i.e. to $\mathtt{a}[i]$ and $\mathtt{a}[i+1]$ - like in the above Max procedure), BSM and flat fragments are inappropriate: you need the fragment of the present contribution (SIL is often - but not always - sufficient as well);
- the peculiar feature of our proposal is modularity: building a library of (even very small or ad hoc) acceleratable fragments, one increases covered applications.

Acceleration for arrays

Suppose now that all loops in our programs can be 'vertically' accelerated and that the resulting $\exists^*\forall$-formulae fall into a decidable fragment.

- This leads to full decidability of safety problems in case the program has a flat control flow (= each control state belongs to at most one cycle).
- Programs consisting on sequences of searching, copying, comparing, etc. functions are of this kind.
- In case the control flow is not flat there is still the possibility of approximating via instantiations the $\exists^*\forall$-formulae arising from accelerations (this is what our tool MCMT does).

Acceleration for arrays

Suppose now that all loops in our programs can be 'vertically' accelerated and that the resulting $\exists^*\forall$-formulae fall into a decidable fragment.

- This leads to full decidability of safety problems in case the program has a flat control flow (= each control state belongs to at most one cycle).
- Programs consisting on sequences of searching, copying, comparing, etc. functions are of this kind.
- In case the control flow is not flat there is still the possibility of approximating via instantiations the $\exists^*\forall$-formulae arising from accelerations (this is what our tool MCMT does).

Acceleration for arrays

Suppose now that all loops in our programs can be 'vertically' accelerated and that the resulting $\exists^*\forall$-formulae fall into a decidable fragment.

- This leads to full decidability of safety problems in case the program has a flat control flow (= each control state belongs to at most one cycle).
- Programs consisting on sequences of searching, copying, comparing, etc. functions are of this kind.
- In case the control flow is not flat there is still the possibility of approximating via instantiations the $\exists^*\forall$-formulae arising from accelerations (this is what our tool MCMT does).

Acceleration for arrays

Suppose now that all loops in our programs can be 'vertically' accelerated and that the resulting $\exists^*\forall$-formulae fall into a decidable fragment.

- This leads to full decidability of safety problems in case the program has a flat control flow (= each control state belongs to at most one cycle).
- Programs consisting on sequences of searching, copying, comparing, etc. functions are of this kind.
- In case the control flow is not flat there is still the possibility of approximating via instantiations the $\exists^*\forall$-formulae arising from accelerations (this is what our tool MCMT does).

Our tools

The results of the present contribution have not been plugged into our tools yet. We can very briefly report about the implementation [ATVA 2014] of results from our previous FroCoS 13 paper.

- MCMT is a large spectrum model checker for array based systems; it makes use of both acceleration and abstraction/refinement techniques;
- Booster is a specialized tool for model checking imperative array programs.

Our tools

The results of the present contribution have not been plugged into our tools yet. We can very briefly report about the implementation [ATVA 2014] of results from our previous FroCoS 13 paper.

- MCMT is a large spectrum model checker for array based systems; it makes use of both acceleration and abstraction/refinement techniques;
- Booster is a specialized tool for model checking imperative array programs.

Our tools

The results of the present contribution have not been plugged into our tools yet. We can very briefly report about the implementation [ATVA 2014] of results from our previous FroCoS 13 paper.

- MCMT is a large spectrum model checker for array based systems; it makes use of both acceleration and abstraction/refinement techniques;
- Booster is a specialized tool for model checking imperative array programs.

Our tools

Booster analyzes the control flow of the program: if it is flat and cycles match syntactic patterns leading to the computation of accelerations, proof obbligations for Z3 are generated in order to decide safety.

In case this strategy does not apply, Booster invoques in parallel many copies of MCMT (with a portfolio policy for different settings of abstraction parameters). Inside abstraction phase, MCMT computes array accelerations and approximates the resulting $\exists^*\forall$-formulae via instantiations.

Endowing MCMT the ability of dealing directly with $\exists^*\forall^*$-formulae belonging to decidable fragments is (nontrivial) future work.

Our tools

Booster analyzes the control flow of the program: if it is flat and cycles match syntactic patterns leading to the computation of accelerations, proof obbligations for Z3 are generated in order to decide safety.

In case this strategy does not apply, Booster invoques in parallel many copies of MCMT (with a portfolio policy for different settings of abstraction parameters). Inside abstraction phase, MCMT computes array accelerations and approximates the resulting $\exists^*\forall$-formulae via instantiations.

Endowing MCMT the ability of dealing directly with $\exists^*\forall^*$-formulae belonging to decidable fragments is (nontrivial) future work.

Our tools

Booster analyzes the control flow of the program: if it is flat and cycles match syntactic patterns leading to the computation of accelerations, proof obbligations for Z3 are generated in order to decide safety.

In case this strategy does not apply, Booster invoques in parallel many copies of MCMT (with a portfolio policy for different settings of abstraction parameters). Inside abstraction phase, MCMT computes array accelerations and approximates the resulting $\exists^*\forall$-formulae via instantiations.

Endowing MCMT the ability of dealing directly with $\exists^*\forall^*$-formulae belonging to decidable fragments is (nontrivial) future work.

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage `www.sanfoundry.com/c-programming-examples-arrays/`.

See Booster's web page for more
`http://verify.inf.usi.ch/booster`

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage `www.sanfoundry.com/c-programming-examples-arrays/`.

See Booster's web page for more
`http://verify.inf.usi.ch/booster`

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage `www.sanfoundry.com/c-programming-examples-arrays/`.

See Booster's web page for more
`http://verify.inf.usi.ch/booster`

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage `www.sanfoundry.com/c-programming-examples-arrays/`.

See Booster's web page for more
`http://verify.inf.usi.ch/booster`

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage www.sanfoundry.com/c-programming-examples-arrays/.

See Booster's web page for more
http://verify.inf.usi.ch/booster

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage `www.sanfoundry.com/c-programming-examples-arrays/`.

See Booster's web page for more
`http://verify.inf.usi.ch/booster`

Booster: experiments

Booster has been evaluated on the following sets of benchmarks:

- standard: programs with arrays used in related works on array analysis;
- sorting: sorting procedures for arrays;
- data_structures: programs exploiting arrays for defining more complex data-structures (e.g., sets);
- svcomp: benchmarks taken from the SV-COMP repository, 'loops' folder;
- sanfoundry: programs taken and adapted from the webpage `www.sanfoundry.com/c-programming-examples-arrays/`.

See Booster's web page for more
`http://verify.inf.usi.ch/booster`

Thank you!