Lazy Abstraction with Interpolants for Arrays

F. Alberti[1], R. Bruttomesso[2], S. Ghilardi[2],
S. Ranise[3], N. Sharygina[1]

[1]Formal Verification and Security Lab, University of Lugano, Switzerland
[2] University of Milan, Italy
[3]FBK-Irst, Trento, Italy

LPAR-18
March 12, 2012

Context

Software model checking:

- Given a program P and a property ϕ, does P exhibit an execution violating ϕ?

Context

Software model checking:

- Given a program P and a property ϕ, does P exhibit an execution violating ϕ?

- Transition-relation representation of input program
- Predicate abstraction [GS97]
- Lazy Abstraction [HJMS02]
  - different degrees of precision for different parts of the program

Several abstraction refinement strategies

Several abstraction refinement strategies

Interpolants from (unsatisfiable) formulas representing infeasible counterexamples [HJMM04, McM06]

1. $\phi = \phi_1 \wedge \cdots \wedge \phi_n$ is satisfiable iff $\pi = \tau_1 \cdots \tau_n$ is feasible
2. Retrieve a set of (quantifier-free) formulas $\{\psi_0, \ldots, \psi_n\}$ s.t.
   - $\psi_0 \equiv \top$
   - $\psi_n \equiv \bot$
   - $\psi_{i-1} \wedge \phi_i \models \psi_i$
   - ψ_i is over the common signature of ϕ_i and ϕ_{i+1}

The presence of arrays complicates the framework because:

The presence of arrays complicates the framework because:

1. Useful array predicates require quantifiers, e.g., "the array a is sorted"

$$\forall i, j.(0 \leq i < j < a.length) \Rightarrow a[i] \leq a[j]$$

The presence of arrays complicates the framework because:

1. Useful array predicates require quantifiers, e.g., "the array a is sorted"
$$\forall i, j.(0 \leq i < j < a.length) \Rightarrow a[i] \leq a[j]$$

2. No quantifier-free interpolation for the "standard" theory of array [KMZ06]

- Our goal: Software model checking of programs handling arrays with (possibly universally quantified) assertions

Context: Software model checking

- Our goal: Software model checking of programs handling arrays with (possibly universally quantified) assertions

- Model Checking Modulo Theories framework [GR10]
  - ✔ Native handling of arrays
  - ✔ Quantified formulas representing set of reachable states

Context: Software model checking

Lazy Abstraction with Interpolants for Arrays

- Our goal: Software model checking of programs handling arrays with (possibly universally quantified) assertions

- Model Checking Modulo Theories framework [GR10]
  - ✔ Native handling of arrays
  - ✔ Quantified formulas representing set of reachable states

⇒ Redefine the interpolation-based Lazy Abstraction approach:
  1. Quantifier handling and Array-reasoning adapted from MCMT
  2. New quantifier-free interpolation algorithm for arrays

Outline

1 Array-based Transition Systems

2 Unwinding Array-based Transition Systems

3 Refinement with Interpolants

4 Completeness

5 Experiments

Outline

1 Array-based Transition Systems

2 Unwinding Array-based Transition Systems

3 Refinement with Interpolants

4 Completeness

5 Experiments

Array-based Transition Systems

Class of guarded assignment systems with array state variables

Array-based Transition Systems

Class of guarded assignment systems with array state variables

- A mono-sorted (`INDEX`) theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes of arrays
- A multi-sorted ($\mathtt{ELEM}_l$) theory $T_E = (\Sigma_E, \mathcal{C}_E)$ for data inside arrays

Array-based Transition Systems

Class of guarded assignment systems with array state variables

- A mono-sorted (`INDEX`) theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes of arrays
- A multi-sorted ($\mathtt{ELEM}_l$) theory $T_E = (\Sigma_E, \mathcal{C}_E)$ for data inside arrays
- A theory $A_I^E = (\Sigma, \mathcal{C})$ linking T_I and T_E

Array-based Transition Systems

Class of guarded assignment systems with array state variables

- A mono-sorted ($\mathtt{INDEX}$) theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes of arrays
- A multi-sorted ($\mathtt{ELEM}_l$) theory $T_E = (\Sigma_E, \mathcal{C}_E)$ for data inside arrays
- A theory $A_I^E = (\Sigma, \mathcal{C})$ linking T_I and T_E
  - sort symbols of A_I^E are $\mathtt{INDEX}, \mathtt{ELEM}_l$ and $\mathtt{ARRAY}_l$
  - $\Sigma = \Sigma_I \cup \Sigma_E \cup \{\_[\_]_l\}_l$
    - $\_[\_]_i : \mathtt{ARRAY}_i \times \mathtt{INDEX} \rightarrow \mathtt{ELEM}_i$

Array-based Transition Systems

Class of guarded assignment systems with array state variables

- A mono-sorted ($\texttt{INDEX}$) theory $T_I = (\Sigma_I, \mathcal{C}_I)$ for indexes of arrays
- A multi-sorted ($\texttt{ELEM}_l$) theory $T_E = (\Sigma_E, \mathcal{C}_E)$ for data inside arrays
- A theory $A_I^E = (\Sigma, \mathcal{C})$ linking T_I and T_E
  - sort symbols of A_I^E are $\texttt{INDEX}, \texttt{ELEM}_l$ and $\texttt{ARRAY}_l$
  - $\Sigma = \Sigma_I \cup \Sigma_E \cup \{\_[\_]_l\}_l$
    - $\_[\_]_i : \texttt{ARRAY}_i \times \texttt{INDEX} \to \texttt{ELEM}_i$
  - $\mathcal{M} \in \mathcal{C}$ iff
    - $\texttt{ARRAY}_l^{\mathcal{M}} = [\texttt{INDEX}^{\mathcal{M}} \to \texttt{ELEM}_l^{\mathcal{M}}]$
    - $\_[\_]_l^{\mathcal{M}}$ is function application
    - $\mathcal{M}|_{\Sigma_I} \in \mathcal{C}_I$ and $\mathcal{M}|_{\Sigma_E} \in \mathcal{C}_E$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

$\mathbf{v} = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$

- $\mathbf{a}$ is a set of variables of sort $\mathtt{ARRAY}_l$
- $\mathbf{c}$ is a set of variables of sort $\mathtt{INDEX}$
- $\mathbf{d}$ is a set of variables of sort $\mathtt{ELEM}_l$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

$\mathbf{v} = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$

- $\mathbf{a}$ is a set of variables of sort $\texttt{ARRAY}_l$
- $\mathbf{c}$ is a set of variables of sort $\texttt{INDEX}$
- $\mathbf{d}$ is a set of variables of sort $\texttt{ELEM}_l$

- $\texttt{pc} \in \mathbf{d}$ ranging over a set L of locations

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

$\mathbf{v} = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$

- $\mathbf{a}$ is a set of variables of sort $\mathtt{ARRAY}_l$
- $\mathbf{c}$ is a set of variables of sort $\mathtt{INDEX}$
- $\mathbf{d}$ is a set of variables of sort $\mathtt{ELEM}_l$

- $\mathtt{pc} \in \mathbf{d}$ ranging over a set L of locations
- $I(\mathbf{v}) \triangleq (\mathtt{pc} = l_I)$ is the initial state of $\mathcal{S}$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

$\mathbf{v} = \{\mathbf{a}, \mathbf{c}, \mathbf{d}\}$

- $\mathbf{a}$ is a set of variables of sort $\texttt{ARRAY}_l$
- $\mathbf{c}$ is a set of variables of sort $\texttt{INDEX}$
- $\mathbf{d}$ is a set of variables of sort $\texttt{ELEM}_l$

- $\texttt{pc} \in \mathbf{d}$ ranging over a set L of locations
- $I(\mathbf{v}) \triangleq (\texttt{pc} = l_I)$ is the initial state of $\mathcal{S}$
- $U(\mathbf{v}) \triangleq (\texttt{pc} = l_E)$ is the error state of $\mathcal{S}$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

The τ_h's are guarded assignments in functional form

$$\exists \underline{k} \begin{pmatrix} \phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix}$$

Array-based Transition Systems

An *array-based system* for T_I, T_E is a pair $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$

The τ_h's are guarded assignments in functional form

$$\exists \underline{k} \begin{pmatrix} \phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix}$$

For all τ_h:

- $\phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \models \mathtt{pc} = l$; $\quad src(\tau_h) = l$
- $\mathtt{pc}' = l'$; $\quad trg(\tau_h) = l'$

Array-based Transition Systems

Translation from source code

```
function find ( int a[ ] , int n ) {
    c = 0;
    while (c < a.length ∧ a[c] ≠ n) c = c + 1;
    if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
        ERROR;
}
```

```
  function find ( int a[ ] , int n ) {
1     c = 0;
2     while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3     if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4        ERROR;
  }
```

$T_I = \mathcal{LIA}$ with a constant $\mathtt{a.length}$

$T_E = \mathcal{LIA}$ with a constant $\mathtt{n}$ $\cup$ $\{1, 2, 3, 4\}$

$\mathbf{a} = \{\mathtt{a}\}$, $\mathbf{c} = \{\mathtt{c}\}$, $\mathbf{d} = \{\mathtt{pc}\}$

Array-based Transition Systems

Translation from source code

function $\texttt{find}$ ($\texttt{int a}[\,]$, $\texttt{int n}$) {

1 $\quad \texttt{c} = 0;$

2 $\quad \texttt{while}\ (\texttt{c} < \texttt{a.length} \wedge \texttt{a}[\texttt{c}] \neq \texttt{n})\ \texttt{c} = \texttt{c} + 1;$

3 $\quad \texttt{if}\ (\ \texttt{c} \geq \texttt{a.length} \wedge \exists x.(x \geq 0 \wedge x < \texttt{a.length} \wedge \texttt{a}[x] = \texttt{n})\ )$

4 $\qquad \texttt{ERROR};$

}

$$l_I = 1 \qquad\qquad l_E = 4$$

Array-based Transition Systems

Translation from source code

```
    function find ( int a[ ] , int n ) {
1       c = 0;
2       while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3       if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4           ERROR;
    }
```

$$\tau_1 \equiv \mathtt{pc} = 1 \wedge \mathtt{pc}' = 2 \wedge \mathtt{c}' = 0$$

Array-based Transition Systems

Translation from source code

```
  function find ( int a[ ] , int n ) {
1    c = 0;
2    while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3    if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4       ERROR;
  }
```

$$\tau_2 \equiv \quad \mathtt{pc} = 2 \wedge \mathtt{c} < \mathtt{a.length} \wedge \quad \mathtt{a}[\mathtt{c}] \neq \mathtt{n} \quad \wedge\, \mathtt{pc}' = 2 \wedge \mathtt{c}' = \mathtt{c} + 1$$

Array-based Transition Systems

Translation from source code

```
   function find ( int a[ ] , int n ) {
1     c = 0;
2     while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3     if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4        ERROR;
   }
```

$$
\begin{aligned}
\tau_2 &\equiv \quad \mathtt{pc} = 2 \wedge \mathtt{c} < \mathtt{a.length} \wedge \quad \mathtt{a}[\mathtt{c}] \neq \mathtt{n} \quad \wedge \mathtt{pc}' = 2 \wedge \mathtt{c}' = \mathtt{c} + 1 \\
&\equiv \exists x. \quad \mathtt{pc} = 2 \wedge \mathtt{c} < \mathtt{a.length} \wedge \quad x = \mathtt{c} \wedge \mathtt{a}[x] \neq \mathtt{n} \quad \wedge \mathtt{pc}' = 2 \wedge \mathtt{c}' = \mathtt{c} + 1
\end{aligned}
$$

Array-based Transition Systems

Translation from source code

```
  function find ( int a[ ] , int n ) {
1   c = 0;
2   while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3   if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4     ERROR;
  }
```

$$\tau_3 \equiv \mathtt{pc} = 2 \land \mathtt{c} \geq \mathtt{a.length} \land \mathtt{pc}' = 3$$

```
   function find ( int a[ ] , int n ) {
1      c = 0;
2      while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3      if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4         ERROR;
   }
```

$$\tau_4 \equiv \exists x.\mathtt{pc} = 2 \wedge x = \mathtt{c} \wedge \mathtt{a}[x] = \mathtt{n} \wedge \mathtt{pc}' = 3$$

Array-based Transition Systems

Translation from source code

```
   function find ( int a[ ] , int n ) {
1      c = 0;
2      while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3      if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4         ERROR;
   }
```

$$\tau_5 \equiv \texttt{pc} = 3 \wedge \texttt{c} \geq \texttt{a.length} \wedge \exists x.\ (x \geq 0 \wedge x < \texttt{a.length} \wedge \texttt{a}[x] = \texttt{n}) \wedge \texttt{pc}' = 4$$

Array-based Transition Systems

An *array-based system* $\mathcal{S} = (\mathbf{v}, \{\tau_h\})$ is *safe* w.r.t. an error state $U(\mathbf{v})$ iff the formulas

$$I(\mathbf{v}^{(n)}) \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(n)}, \mathbf{v}^{(n-1)}) \right) \wedge \cdots \wedge \left(\bigvee_h \tau_h(\mathbf{v}^{(1)}, \mathbf{v}^{(0)}) \right) \wedge U(\mathbf{v}^{(0)})$$

are A_I^E-unsatisfiable for every $n \geq 0$

Outline

1. Array-based Transition Systems
2. Unwinding Array-based Transition Systems
3. Refinement with Interpolants
4. Completeness
5. Experiments

Labeled unwinding

A *labeled unwinding* of $\mathcal{S} = \langle \mathbf{v}; \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h \rangle$ is a quadruple (V, E, M_E, M_V)

- (V, E) is a finite rooted tree (let ε be the root)
- M_E, M_V are labeling functions for edges and vertices, respectively

(i) $M_V(\varepsilon) = U(\mathbf{v})$

(ii) $M_V(v)$ is a qff of the kind $\psi(\underline{i}, \mathbf{a}[\underline{i}], \mathbf{c}, \mathbf{d})$ s.t. $M_V(v) \models_{A_I^E} \mathtt{pc} = l$

(iii) $M_E(v, w)$ is the matrix of some $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$ and

- $M_V(w) \models_{A_I^E} \mathtt{pc} = trg(\tau)$
- $M_V(v) \models_{A_I^E} \mathtt{pc} = src(\tau)$
- $M_E(v, w)(\mathbf{v}, \mathbf{v}') \wedge M_V(w)(\mathbf{v}') \models_{A_I^E} M_V(v)(\mathbf{v})$

(iv) for each $\tau \in \{\tau_h(\mathbf{v}, \mathbf{v}')\}_h$ and every non-leaf vertex $w \in V$ s.t. $M_V(w) \models_{A_I^E} pc = trg(\tau)$, there exist $v \in V$ and $(v, w) \in E$ such that $M_E(v, w)$ is the matrix of τ

Unwinding Array-based Transition Systems

EXPAND procedure

Theorem ([GR10])

If τ is in functional form and $M_V(v)$ is an $\exists^I$-formula[a]*, the pre-image of $M_V(v)$ w.r.t. $\tau(\mathbf{v}, \mathbf{v}')$*

$$Pre(\ M_V(v)\ ,\ \tau\ )\ \ \triangleq\ \ \exists \mathbf{v}'.\ (\ \tau(\mathbf{v}, \mathbf{v}')\ \wedge\ M_V(v)(\mathbf{v}')\ )$$

is A_I^E-equivalent to an effectively computable $\exists^I$-formula

[a] A formula of the kind $\exists \underline{k}.\phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d})$, where $\underline{k}$ have sort `INDEX`

Theorem ([GR10])

If τ is in functional form and $M_V(v)$ is an $\exists^I$-formula[a], the pre-image of $M_V(v)$ w.r.t. $\tau(\mathbf{v}, \mathbf{v}')$

$$Pre(\ M_V(v)\ ,\ \tau\ )\ \ \triangleq\ \ \exists \mathbf{v}'.\ (\ \tau(\mathbf{v}, \mathbf{v}')\ \wedge\ M_V(v)(\mathbf{v}')\ )$$

is A_I^E-equivalent to an effectively computable $\exists^I$-formula

[a] A formula of the kind $\exists \underline{k}.\phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d})$, where $\underline{k}$ have sort `INDEX`

1. Fix the format of the formulas we need to handle
2. It can be shown that the pre-image has all the `INDEX` variables of the starting formula

Unwinding Array-based Transition Systems

Complete unwinding

A label unwinding (V, E, M_V, M_E) is *complete* iff there exists a covering, i.e., a set of non-leaf vertexes C s.t.

- $\varepsilon \in C$
- for every $v \in V$ and $(v', v) \in E$, C covers v', i.e.

$$M_V(v')^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}$$

Unwinding Array-based Transition Systems

Complete unwinding

A label unwinding (V, E, M_V, M_E) is *complete* iff there exists a covering, i.e., a set of non-leaf vertexes C s.t.

- $\varepsilon \in C$
- for every $v \in V$ and $(v', v) \in E$, C covers v', i.e.

$$M_V(v')^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}$$

Safe unwinding

A label unwinding (V, E, M_V, M_E) is *safe* iff for all $v \in V$, if $M_V(v) \models_{A_I^E} \mathtt{pc} = l_I$ then $M_V(v)$ is A_I^E-unsatisfiable

- (V, E, M_V, M_E) is not complete
- exists $v \in V$ with a consistent $M_V(v)$, $M_V(v) \models_{A_I^E} \texttt{pc} = l_I$

- (V, E, M_V, M_E) is not complete
- exists $v \in V$ with a consistent $M_V(v)$, $M_V(v) \models_{A_I^E} \mathtt{pc} = l_I$

Given the path $v = v_0 \xrightarrow{\tau_m} v_1 \xrightarrow{\tau_{m-1}} \cdots \xrightarrow{\tau_1} v_m = \varepsilon$, consider the formula

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \tag{1}$$

- (V, E, M_V, M_E) is not complete
- exists $v \in V$ with a consistent $M_V(v)$, $M_V(v) \models_{A_I^E} \texttt{pc} = l_I$

Given the path $v = v_0 \xrightarrow{\tau_m} v_1 \xrightarrow{\tau_{m-1}} \cdots \xrightarrow{\tau_1} v_m = \varepsilon$, consider the formula

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \tag{1}$$

- If (1) is A_I^E-satisfiable, return with UNSAFE

- (V, E, M_V, M_E) is not complete
- exists $v \in V$ with a consistent $M_V(v)$, $M_V(v) \models_{A_I^E} \texttt{pc} = l_I$

Given the path $v = v_0 \xrightarrow{\tau_m} v_1 \xrightarrow{\tau_{m-1}} \cdots \xrightarrow{\tau_1} v_m = \varepsilon$, consider the formula

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \land \cdots \land \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)}) \tag{1}$$

- If (1) is A_I^E-satisfiable, return with UNSAFE
- If (1) is A_I^E-unsatisfiable, retrieve interpolants for $v_0, \ldots, v_m$ to exclude the infeasible counterexample from the model

Outline

1 Array-based Transition Systems

2 Unwinding Array-based Transition Systems

3 Refinement with Interpolants

4 Completeness

5 Experiments

Theorem

If T_I- and T_E-satisfiability is decidable, the A_I^E-satisfiability of formulas like

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)})$$

is decidable.

[1] $\mathbf{a}' = \lambda j.G(\ldots)$ is equivalent to $\forall j.\mathbf{a}'[j] = G(\ldots)$.

Theorem

If T_I- and T_E-satisfiability is decidable, the A_I^E-satisfiability of formulas like

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)})$$

is decidable.

Counterexample made by the conjunction of formulas of the kind

$$\exists \underline{k} \begin{pmatrix} \phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix}$$

[1] $\mathbf{a}' = \lambda j.G(\ldots)$ is equivalent to $\forall j.\mathbf{a}'[j] = G(\ldots)$.

Theorem

If T_I- and T_E-satisfiability is decidable, the A_I^E-satisfiability of formulas like

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)})$$

is decidable.

Counterexample made by the conjunction of formulas of the kind

$$\exists \underline{k} \begin{pmatrix} \phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix}$$

1. Skolemize $\underline{k}$

[1] $\mathbf{a}' = \lambda j.G(\ldots)$ is equivalent to $\forall j.\mathbf{a}'[j] = G(\ldots)$.

Refinement

Checking satisfiability of the counterexample

Theorem

If T_I- and T_E-satisfiability is decidable, the A_I^E-satisfiability of formulas like

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \land \cdots \land \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)})$$

is decidable.

Counterexample made by the conjunction of formulas of the kind

$$\exists \underline{k} \begin{pmatrix} \phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \land \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \land \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \land \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix}$$

1. Skolemize $\underline{k}$
2. (Selective) Instantiation of j with Skolem constants[1]

[1] $\mathbf{a}' = \lambda j.G(\ldots)$ is equivalent to $\forall j.\mathbf{a}'[j] = G(\ldots)$.

Refinement

Checking satisfiability of the counterexample

Theorem

If T_I- and T_E-satisfiability is decidable, the A_I^E-satisfiability of formulas like

$$\tau_1(\mathbf{v}^{(0)}, \mathbf{v}^{(1)}) \wedge \cdots \wedge \tau_m(\mathbf{v}^{(m-1)}, \mathbf{v}^{(m)})$$

is decidable.

Counterexample made by the conjunction of formulas of the kind

$$\exists \underline{k} \begin{pmatrix} \phi(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \\ \mathbf{a}' = \lambda j.\, G(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}, j, \mathbf{a}[j]) \wedge \\ \mathbf{c}' = H(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \wedge \mathbf{d}' = K(\underline{k}, \mathbf{a}[\underline{k}], \mathbf{c}, \mathbf{d}) \end{pmatrix}$$

1. Skolemize $\underline{k}$
2. (Selective) Instantiation of j with Skolem constants[1]
3. Propagate equalities and exploit decision procedures for T_I and T_E

[1] $\mathbf{a}' = \lambda j.G(\ldots)$ is equivalent to $\forall j.\mathbf{a}'[j] = G(\ldots)$.

Refinement

Retrieving interpolants

Theorem

Given an A_I^E-unsatisfiable quantifier-free formula $\psi_1 \wedge \psi_2$, if

- *T_I and T_E admit quantifier-free interpolation algorithms*
- *all* `INDEX` *variables in ψ_2 under the scope of $\_[\_]$ occur also in ψ_1*

Then, there exists a quantifier-free formula ψ_0 such that:

(i) $\psi_2 \models_{A_I^E} \psi_0$

(ii) *$\psi_0 \wedge \psi_1$ is A_I^E-unsatisfiable*

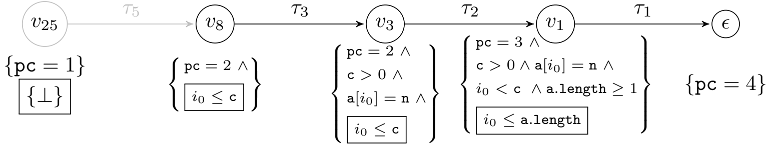
(iii) *all free variables occurring in ψ_0 occur both in ψ_1 and ψ_2*

```
function find ( int a[ ] , int n ) {
1     c = 0;
2     while (c < a.length ∧ a[c] ≠ n) c = c + 1;
3     if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
4        ERROR;
}
```

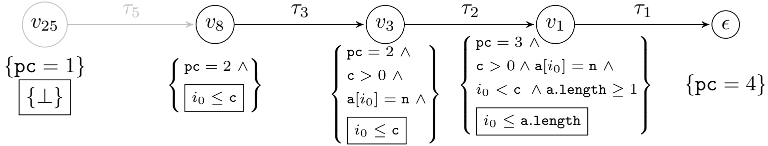
$$v_{25} \xrightarrow{\tau_5} v_8 \xrightarrow{\tau_3} v_3 \xrightarrow{\tau_2} v_1 \xrightarrow{\tau_1} \epsilon$$

$$\{\mathtt{pc} = 1\} \qquad \{\mathtt{pc} = 2\} \qquad \left\{ \begin{array}{l} \mathtt{pc} = 2 \wedge \\ \mathtt{c} > 0 \wedge \\ \mathtt{a}[i_0] = \mathtt{n} \end{array} \right\} \qquad \left\{ \begin{array}{l} \mathtt{pc} = 3 \wedge \\ \mathtt{c} > 0 \wedge \mathtt{a}[i_0] = \mathtt{n} \wedge \\ i_0 < \mathtt{c} \wedge \mathtt{a.length} \geq 1 \end{array} \right\} \qquad \{\mathtt{pc} = 4\}$$

```
function find ( int a[ ] , int n ) {
    c = 0;
    while (c < a.length ∧ a[c] ≠ n) c = c + 1;
    if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
        ERROR;
}
```



```
function find ( int a[ ] , int n ) {
    c = 0;
    while (c < a.length ∧ a[c] ≠ n) c = c + 1;
    if ( c ≥ a.length ∧ ∃x.(x ≥ 0 ∧ x < a.length ∧ a[x] = n) )
        ERROR;
}
```



$$(\mathtt{pc} = 3 \wedge \mathtt{c} > 0 \wedge \mathtt{a.length} \geq 1) \Rightarrow \forall i.\ ((i < \mathtt{c} \wedge i \leq \mathtt{a.length}) \Rightarrow \mathtt{a}[i] \neq \mathtt{n})$$

Outline

1 Array-based Transition Systems

2 Unwinding Array-based Transition Systems

3 Refinement with Interpolants

4 Completeness

5 Experiments

Completeness

Covering set C: for every $v \in V$ and $(v', v) \in E$,

$$M_V(v')^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}$$

Completeness

Covering set C: for every $v \in V$ and $(v', v) \in E$,

$$M_V(v')^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}$$

Computation of the covering set C can be reduced to repeatedly check the A_I^E-unsatisfiability of $\exists^{A,I}\forall^{I}$-sentences.

Completeness

Covering set C: for every $v \in V$ and $(v', v) \in E$,

$$M_V(v')^{\exists} \models_{A_I^E} \bigvee_{w \in C} M_V(w)^{\exists}$$

Computation of the covering set C can be reduced to repeatedly check the A_I^E-unsatisfiability of $\exists^{A,I}\forall^{I}$-sentences.

Theorem ([GR10])

If Σ_I does not contain function symbols and $\mathcal{C}_I$ is closed under substructures, the A_I^E-satisfiability of formulas of the form

$$\exists \mathbf{a}\ \exists \mathbf{c}\ \exists \mathbf{d}\ \exists \underline{i}\ \forall \underline{j}\ \ \psi(\underline{i}, \underline{j}, \mathbf{a}[\underline{i}], \mathbf{a}[\underline{j}], \mathbf{c}, \mathbf{d})$$

is decidable, where ψ is a quantifier-free $\Sigma_I \cup \Sigma_E$-formula.

Completeness (and Termination)
Discussion

- Hypothesis ensuring the completeness of the *covering* test are rather restrictive
  - Those ensuring termination are even more restrictive

Completeness (and Termination)

Discussion

- Hypothesis ensuring the completeness of the *covering* test are rather restrictive
  - Those ensuring termination are even more restrictive

- If they are not met, UNWIND is incomplete, but still sound
  - Only termination is involved

Completeness (and Termination)

Discussion

- Hypothesis ensuring the completeness of the *covering* test are rather restrictive
  - Those ensuring termination are even more restrictive

- If they are not met, UNWIND is incomplete, but still sound
  - Only termination is involved

- What's the behavior of UNWIND in practice (even on systems not meeting termination hypothesis)?

Outline

1. Array-based Transition Systems
2. Unwinding Array-based Transition Systems
3. Refinement with Interpolants
4. Completeness
5. Experiments

Experiments

- SAFARI (SMT-based Abstraction For Arrays with Interpolants) - `http://www.verify.inf.usi.ch/safari`
  - Integration with OpenSMT for SMT-solving

- Experiments over imperative programs handling arrays

Benchmark	Time (s)	Nodes	SMT-calls	Iter.	Result
find (v1)	0.3	5	192	3	SAFE
find (v2)	0.07	5	48	1	SAFE
initialization	0.1	5	96	1	SAFE
max in array	0.9	72	1192	8	SAFE
partition	0.08	20	62	0	SAFE
strcmp	0.4	14	329	4	SAFE
strcpy	0.03	3	15	0	SAFE
vararg	0.03	5	17	0	SAFE
integers	0.02	5	19	0	SAFE
init and test	0.3	27	375	3	SAFE
binary sort	0.3	48	457	2	SAFE
selection sort	0.6	15	478	4	SAFE

Intel i7 @2.66 GHz, equipped with 4GB of RAM and running OSX 10.7

More examples on `http://verify.inf.usi.ch/safari`

Related work

- Ghost variables [FQ02]
- Index predicates [LB07]
- Range predicates [JM07]
- Theorem prover based [KV09, HKV11]
- Abstract interpretation [CCL11, HP08, DDA10]

Conclusions

- A new framework for software model checking of programs with arrays:

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
  2. Lazy-abstraction

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
  2. Lazy-abstraction
  3. New interpolation algorithm

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
  2. Lazy-abstraction
  3. New interpolation algorithm
  4. Efficient (all times below 1 second)

Conclusions

- A new framework for software model checking of programs with arrays:
    1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
    2. Lazy-abstraction
    3. New interpolation algorithm
    4. Efficient (all times below 1 second)
- Future work

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
  2. Lazy-abstraction
  3. New interpolation algorithm
  4. Efficient (all times below 1 second)
- Future work
  - New powerful heuristics for “tuning” interpolation and help convergence (currently under submission)

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
  2. Lazy-abstraction
  3. New interpolation algorithm
  4. Efficient (all times below 1 second)
- Future work
  - New powerful heuristics for "tuning" interpolation and help convergence (currently under submission)
  - Use SAFARI as invariant generator

Conclusions

- A new framework for software model checking of programs with arrays:
  1. Backward reachability, natural handling of quantified predicates for unbounded arrays (MCMT)
  2. Lazy-abstraction
  3. New interpolation algorithm
  4. Efficient (all times below 1 second)
- Future work
  - New powerful heuristics for "tuning" interpolation and help convergence (currently under submission)
  - Use SAFARI as invariant generator
  - Other classes of systems (e.g., distributed algorithms)

Thank you!
Questions?

francesco.alberti@usi.ch

www.inf.usi.ch/phd/alberti

References I

P. Cousot, R. Cousot, and F. Logozzo.
A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis.
In *POPL*, 2011.

I. Dillig, T. Dillig, and A. Aiken.
Fluid Updates: Beyond Strong vs. Weak Updates.
In *Programming Languages and Systems*. 2010.

C. Flanagan and S. Qadeer.
Predicate abstraction for software verification.
In *POPL*, pages 191–202, 2002.

S. Ghilardi and S. Ranise.
Backward Reachability of Array-based Systems by SMT solving: Termination and Invariant Synthesis.
LMCS, 6(4), 2010.

References II

- Susanne Graf and Hassen Saïdi.
  Construction of abstract state graphs with pvs.
  In Orna Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

- Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan.
  Abstractions from proofs.
  In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 232–244. ACM, 2004.

- T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre.
  Lazy Abstraction.
  In *POPL*, 2002.

References III

K. Hoder, L. Kovács, and A. Voronkov.
Invariant Generation in Vampire.
In *TACAS*, pages 60–64, 2011.

N. Halbwachs and Mathias P.
Discovering Properties about Arrays in Simple Programs.
In *PLDI'08*, pages 339–348, 2008.

R. Jhala and K. McMillan.
Array Abstractions from Proofs.
In *CAV*, 2007.

D. Kapur, R. Majumdar, and C. Zarba.
Interpolation for Data Structures.
In *SIGSOFT'06/FSE-14*, pages 105–116, 2006.

L. Kovács and A. Voronkov.
Finding Loop Invariants for Programs over Arrays Using a Theorem Prover.
In *FASE*, pages 470–485, 2009.

S. Lahiri and R. Bryant.
Predicate Abstraction with Indexed Predicates.
TOCL, 9(1), 2007.

Kenneth L. McMillan.
Lazy abstraction with interpolants.
In Thomas Ball and Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.