

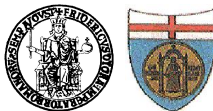
Model Checking Linear Programs with Arrays

A. Armando¹, M. Benerecetti², J. Mantovani¹

¹Artificial Intelligence Laboratory, University of Genova

²Section of Computer Science, University of Napoli "Federico II"

Edinburgh, 11/07/05



Abstract, Check, Refine: the CEGAR approach

- Counterexample-guided abstraction refinement is one of the leading approaches to software model checking;
- In this context, **Boolean Programs** have been proposed (e.g. in SLAM and BLAST), as possible abstractions of imperative programs.

Abstract, Check, Refine: the CEGAR approach

- Counterexample-guided abstraction refinement is one of the leading approaches to software model checking;
- In this context, **Boolean Programs** have been proposed (e.g. in SLAM and BLAST), as possible abstractions of imperative programs.

Boolean Programs, are imperative programs such that

- they have the same control flow graph as the corresponding concrete programs they abstract;
- their variables are restricted to range over boolean values only;
- each boolean variable is in a one-to-one relation with a predicate of the concrete program.

- The detection of a spurious execution trace leads to a new iteration of the check-and-refine loop.
- The efficiency of the approach depends in a critical way on the number of spurious execution traces allowed by the abstract program.
- The closer is the abstraction to the original program the smaller is the number of spurious execution traces that may be necessary to analyse.

Linear Programs

In a previous paper ([Armando et al., ICFEM04]) we proposed **Linear Programs** as a finer grained model for imperative programs and showed how the model checking procedure used in SLAM can be generalised to a model checking procedure for Linear Programs.

Linear Programs

In a previous paper ([Armando et al., ICFEM04]) we proposed **Linear Programs** as a finer grained model for imperative programs and showed how the model checking procedure used in SLAM can be generalised to a model checking procedure for Linear Programs.

Linear Programs

Linear Programs differ from boolean programs in that:

- program variables can range over a numeric domain \mathcal{D} (e.g. \mathbb{Z} or \mathbb{R});
- all conditions and assignments to variables involve linear expressions, i.e. expressions of the form

$$c_0 + c_1x_1 + \dots + c_nx_n,$$

where c_0, \dots, c_n are numeric constants in \mathbb{Z} and x_1, \dots, x_n are program variables ranging over \mathcal{D} .

- Our current work deals with the extension of our model checking procedure to support linear programs featuring:
 - a symbol for *undefined* values ($*$),
 - conditional expressions ($e_1 ? e_2 : e_3$).
- This extension is particularly important as it paves the way to the construction of model checking procedures for wider classes of imperative programs such as, e.g., **Linear Programs with Arrays**.
- We show an abstraction method from Linear Programs with Arrays to Linear Programs.

Abstracting Linear Programs with Arrays

Let a be an array, and let $R(a) = \{k_1, \dots, k_n\}$ be a set of array indexes.

Abstracting Linear Programs with Arrays

Let a be an array, and let $R(a) = \{k_1, \dots, k_n\}$ be a set of array indexes.

Expressions \rightarrow Conditional Expressions with *undefined*

$$\begin{array}{c} a[e] \\ \downarrow \\ (e == k_1 ? a_{k_1} : (e == k_2 ? a_{k_2} : \dots (e == k_n ? a_{k_n} : *) \dots)) \end{array}$$

Abstracting Linear Programs with Arrays

Let a be an array, and let $R(a) = \{k_1, \dots, k_n\}$ be a set of array indexes.

Expressions \rightarrow Conditional Expressions with *undefined*

$$\begin{array}{c} a[e] \\ \downarrow \\ (e == k_1 ? a_{k_1} : (e == k_2 ? a_{k_2} : \dots (e == k_n ? a_{k_n} : *) \dots)) \end{array}$$

Assignments \rightarrow Parallel Assignments

$$\begin{array}{c} a[e_1] = e_2; \\ \downarrow \\ a_{k_1}, \dots, a_{k_n} = (e_1 == k_1 ? e_2 : a_{k_1}), \dots, (e_1 == k_n ? e_2 : a_{k_n}); \end{array}$$

where every a_{k_i} models the $(k_i)^{th}$ array element.

(If $n = 0$, the assignment above reduces to a skip (;) statement.)

Original Program

```
1  int i , a [30];
2  void main(){
3      a[1] = 1;
4      i=0;
5      while (a[i]!=1&& i <30){
6          a[i] = 2*i;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Question

Is line 10 reachable?

Original Program

```
1  int i , a [30];
2  void main () {
3      a [1] = 1;
4      i = 0;
5      while ( a [ i ] != 1 && i < 30 ) {
6          a [ i ] = 2 * i;
7          i = i + 1;
8      }
9      if ( i > 1 )
10     ERROR : ;
11 }
```

Question

Is line 10 reachable?

Method

We abstract every occurrence of array elements

Example

Original Program

```
1  int i , a [30];
2  void main(){
3      a[1] = 1;
4      i=0;
5      while (a[i]!=1&& i <30){
6          a[i] = 2*i;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Abstraction wrt. $R(a) = \emptyset$

```
1  int i;
2  void main(){
3      ;
4      i=0;
5      while (*!=1&& i <30){
6          ;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Question

Is line 10 reachable?

Abstraction wrt. $R(a) = \emptyset$

```
1  int i;  
2  void main(){  
3      ;  
4      i=0;  
5      while(*!=1&& i < 30){  
6          ;  
7          i = i+1;  
8      }  
9      if (i > 1)  
10     ERROR: ;  
11 }
```

Example

Question

Is line 10 reachable?

Answer

Yes! The trace is:

3, 4, 5, 6, 7, ..., 5, 6, 7, 5, 9, 10

Abstraction wrt. $R(a) = \emptyset$

```
1  int i;  
2  void main(){  
3      ;  
4      i=0;  
5      while(*!=1&& i < 30){  
6          ;  
7          i = i+1;  
8      }  
9      if (i > 1)  
10     ERROR: ;  
11 }
```

Trace Simulation

- The simulation step builds a formula ϕ in the combined theory of linear arithmetics + arrays s.t. $SAT(\phi) \leftrightarrow$ the trace is feasible.
- We use CVC Lite to check for satisfiability.
- CVC Lite returns that $UNSAT(\phi)$.
- By inspecting the unsatisfiable core it can be determined that this is due to array index 1.

Original Program

```
1  int i , a [ 30 ] ;
2  void main () {
3      a [ 1 ] = 1 ;
4      i = 0 ;
5      while ( a [ i ] != 1 && i < 30 ) {
6          a [ i ] = 2 * i ;
7          i = i + 1 ;
8      }
9      if ( i > 1 )
10     ERROR : ;
11 }
```

Trace Simulation

- The simulation step builds a formula ϕ in the combined theory of linear arithmetics + arrays s.t. $SAT(\phi) \leftrightarrow$ the trace is feasible.
- We use CVC Lite to check for satisfiability.
- CVC Lite returns that $UNSAT(\phi)$.
- By inspecting the unsatisfiable core it can be determined that this is due to array index 1.

Refinement

We add the index 1 to $R(a)$.

Original Program

```
1  int i, a[30];
2  void main(){
3      a[1] = 1;
4      i=0;
5      while (a[i]!=1&& i < 30){
6          a[i] = 2*i;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Example

Abstraction wrt. $R(a) = \{1\}$

```
1  int i, a1;
2  void main(){
3      a1=(1==1)?1:*;
4      i=0;
5      while (((i==1)?a1:*)!=1)&& i < 30){
6          a1=(i==1)?2*i:a1;
7          i = i+1;
8      }
9      if(i > 1)
10     ERROR: ;
11 }
```

Original Program

```
1  int i, a[30];
2  void main(){
3      a[1] = 1;
4      i=0;
5      while (a[i]!=1&&i < 30){
6          a[i] = 2*i;
7          i = i+1;
8      }
9      if(i > 1)
10     ERROR: ;
11 }
```

Example

Abstraction wrt. $R(a) = \{1\}$

```
1  int i, a1;
2  void main(){
3      a1=(1==1)?1:*;
4      i=0;
5      while (((i==1)?a1:*)!=1)&& i < 30){
6          a1=(i==1)?2*i : a1;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Question

Is line 10 reachable in the refined program?

Example

Abstraction wrt. $R(a) = \{1\}$

```
1  int i, a1;
2  void main(){
3      a1=(1==1)?1:*;
4      i=0;
5      while (((i==1)?a1:*)!=1)&& i < 30){
6          a1=(i==1)?2*i : a1;
7          i = i+1;
8      }
9      if (i > 1)
10     ERROR: ;
11 }
```

Question

Is line 10 reachable in the refined program?

Final result

No! The refined abstract program is safe, and so is the concrete one.

The Model Checking Procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv), the procedure computes reachability of vertices of the CFG, using:

- **Path Edges** to represent the reachability status of vertices,
- **Summary Edges** to record the input/output behaviour of procedures,

Path Edges and Summary Edges are represented by means of **Abstract Disjunctive Linear Constraints** (ADLCs).

The Model Checking Procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv), the procedure computes reachability of vertices of the CFG, using:

- **Path Edges** to represent the reachability status of vertices,
- **Summary Edges** to record the input/output behaviour of procedures,

Path Edges and Summary Edges are represented by means of **Abstract Disjunctive Linear Constraints** (ADLCs).

Let V be the set of program variables. Let U be an infinite set of variables such that $U \cap V = \emptyset$. An ADLC is an expression of the form $\lambda \mathbf{x}. \lambda \mathbf{x}'. \bigvee_i \bigwedge_j \exists U. c_{ij}$, and \mathbf{x} is an n -tuple comprising the free variables occurring in D .

The Model Checking Procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv), the procedure computes reachability of vertices of the CFG, using:

- **Path Edges** to represent the reachability status of vertices,
- **Summary Edges** to record the input/output behaviour of procedures,

Path Edges and Summary Edges are represented by means of **Abstract Disjunctive Linear Constraints** (ADLCs).

Let V be the set of program variables. Let U be an infinite set of variables such that $U \cap V = \emptyset$. An ADLC is an expression of the form $\lambda \mathbf{x}. \lambda \mathbf{x}'. \bigvee_i \bigwedge_j \exists U. c_{ij}$, and \mathbf{x} is an n -tuple comprising the free variables occurring in D .

- For each node i in the CFG of the program, our procedure incrementally builds a **symbolic representation** of the set of path edges incident in i .
- Reachability of a statement s_i then boils down to checking whether the set of path edges incident in i is not empty.

The Model Checking Procedure

Similarly to the SLAM's Bebop approach (based on the Interprocedural Dataflow Analysis procedure of Reps, Horwitz, Sagiv), the procedure computes reachability of vertices of the CFG, using:

- **Path Edges** to represent the reachability status of vertices,
- **Summary Edges** to record the input/output behaviour of procedures,

The Model Checking Procedure - Conditional Expressions

We define the relation $e \rightarrow (B, ne)$ to be the smallest relation such that $e \rightarrow (\emptyset, e)$ for all linear expressions e and closed under the following inference rules:

$$\frac{e_1 \rightarrow (B_1, ne_1) \quad e_1 \rightarrow (B_2, ne_2)}{(e_1 \text{ op } e_2) \rightarrow (B_1 \cup B_2, (ne_1 \text{ op } ne_2))} \text{ if } \text{op} \in \{*, +, >=, =, <, <, >, ==, !=\}$$

$$\frac{b \rightarrow (B, nb) \quad e_1 \rightarrow (B_1, ne_1)}{(b ? e_1 : e_2) \rightarrow (B \cup B_1 \cup \{nb^+\}, ne_1)}$$

$$\frac{b \rightarrow (B, nb) \quad e_2 \rightarrow (B_2, ne_2)}{(b ? e_1 : e_2) \rightarrow (B \cup B_2 \cup \{nb^-\}, ne_2)}$$

where b^+ (b^-) is $b \neq 0$ ($b == 0$, resp.) if b is a linear expression and b ($! b$, resp.) if b is a boolean expression.

The Model Checking Procedure - Conditional Expressions

We define the relation $e \rightarrow (B, ne)$ to be the smallest relation such that $e \rightarrow (\emptyset, e)$ for all linear expressions e and closed under the following inference rules:

$$\frac{e_1 \rightarrow (B_1, ne_1) \quad e_1 \rightarrow (B_2, ne_2)}{(e_1 \text{ op } e_2) \rightarrow (B_1 \cup B_2, (ne_1 \text{ op } ne_2))} \text{ if } \text{op} \in \{*, +, >=, =<, <, >, ==, !=\}$$

$$\frac{b \rightarrow (B, nb) \quad e_1 \rightarrow (B_1, ne_1)}{(b ? e_1 : e_2) \rightarrow (B \cup B_1 \cup \{nb^+\}, ne_1)} \quad \frac{b \rightarrow (B, nb) \quad e_2 \rightarrow (B_2, ne_2)}{(b ? e_1 : e_2) \rightarrow (B \cup B_2 \cup \{nb^-\}, ne_2)}$$

where b^+ (b^-) is $b \neq 0$ ($b == 0$, resp.) if b is a linear expression and b ($! b$, resp.) if b is a boolean expression.

It can be shown that if $e \rightarrow (B, ne)$, then both the linear expression ne and the set B of linear expressions do not contain conditionals.

The Model Checking Procedure - Undefined Symbol

- Let U be an infinite set of fresh variables.
- If e is a linear expression, by e^* we indicate the expression obtained from e by replacing every occurrence of $*$ in e with a distinct variable u_j in U .
- Then we can **quantify** the variables u_j in e^* :

$$\exists u_1 \dots \exists u_k. e^*$$

The Model Checking Procedure - Undefined Symbol

- Let U be an infinite set of fresh variables.
- If e is a linear expression, by e^* we indicate the expression obtained from e by replacing every occurrence of $*$ in e with a distinct variable u_j in U .
- Then we can **quantify** the variables u_j in e^* :

$$\exists u_1 \dots \exists u_k . e^*$$

Notice that if e is a linear expression without conditionals, then $\exists u_1 \dots \exists u_k . e^*$ can be always simplified to an equivalent Linear Arithmetics expression.

The Model Checking Procedure - Example

- For assignments we define

$$\alpha(z, e) = \bigvee \left\{ \exists U. (z = ne \wedge \bigwedge B)^* : e \rightarrow (B, ne) \right\}.$$

The Model Checking Procedure - Example

- For assignments we define

$$\alpha(z, e) = \bigvee \left\{ \exists U. (z = ne \wedge \bigwedge B)^* : e \rightarrow (B, ne) \right\}.$$

- Let $z = e$ be an assignment where

$$e = (3x + (y < 0 ? * : y)).$$

The Model Checking Procedure - Example

- For assignments we define

$$\alpha(z, e) = \bigvee \left\{ \exists U. (z = ne \wedge \bigwedge B)^* : e \rightarrow (B, ne) \right\}.$$

- Let $z = e$ be an assignment where

$$e = (3x + (y < 0 ? * : y)).$$

- we get rid of conditionals:

$$e \rightarrow (\{y < 0\}, 3x + *) \quad e \rightarrow (\{\neg y < 0\}, 3x + y)$$

The Model Checking Procedure - Example

- For assignments we define

$$\alpha(z, e) = \bigvee \left\{ \exists U. (z = ne \wedge \bigwedge B)^* : e \rightarrow (B, ne) \right\}.$$

- Let $z = e$ be an assignment where

$$e = (3x + (y < 0 ? * : y)).$$

- we get rid of conditionals:

$$e \rightarrow (\{y < 0\}, 3x + *) \quad e \rightarrow (\{\neg y < 0\}, 3x + y)$$

- we then encode the assignment as:

$$\alpha(z, e) = (\exists u_1. (y < 0 \wedge z = 3x + u_1) \vee (\neg(y < 0) \wedge z = 3x + y))$$

The Model Checking Procedure

Theorem

*The model checking procedure for linear programs extended with the * symbol and the conditional expressions is sound and complete.*

- We have extended the previously implemented model checker EUREKA so that it now supports the analysis of linear programs extended with the $*$ symbol and conditional expressions.
- Entailment checking is done by using ICS v2.0 as a decision procedure for the boolean combination of linear arithmetic constraints.
- In order to assess the effectiveness of the approach we have also developed in EUREKA a prototype implementation of the counterexample-guided abstraction refinement procedure outlined in this talk.
- feasibility checks performed using CVC Lite as it is a complete prover for the union of the theory of arrays and linear arithmetics.

Experimental Results

- The programs below make use of arrays and arithmetics.
- They have been used as regression tests for EUREKA.

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

Experimental Results

- The programs below make use of arrays and arithmetics.
- They have been used as regression tests for EUREKA.

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

Experimental Results

- The programs below make use of arrays and arithmetics.
- They have been used as regression tests for EUREKA.

Program	EUREKA	BLAST	CBMC
array_init.c	safe	error	safe
array_init_assign.c	safe	unsafe	safe
complex_guard.c	safe	unsafe	safe
simple_swap.c	safe	safe	safe
sequential_swap_call.c	safe	error	safe
simple_array_inversion.c	safe	unsafe	safe
bubblesort_inner_loop.c	unsafe	error	unsafe
bubblesort.c	unsafe	safe	unsafe
array_max.c	safe	error	safe
wrong_loop.c	unsafe	error	mout
loop_on_input.c	unsafe	unsafe	mout
simple_control_on_input.c	unsafe	unsafe	mout

CBMC

- CBMC encodes the (bounded) verification problem for C programs into a SAT formula.
- it uses a “physical” model of memory: every integer variable is modelled as a word of n bits, where n is either 8, 16, 32, or 64.
- it finds proper *unwinding assertions* in order to automatically determine a safe bound for model checking the program. **The bound can not always be found automatically.**

CBMC

- CBMC encodes the (bounded) verification problem for C programs into a SAT formula.
- it uses a “physical” model of memory: every integer variable is modelled as a word of n bits, where n is either 8, 16, 32, or 64.
- it finds proper *unwinding assertions* in order to automatically determine a safe bound for model checking the program. **The bound can not always be found automatically.**

BLAST

- performs (lazy) boolean abstraction and refinement
- models any “[...] expression $p + i$, where p is a pointer and i is an integer, as yielding a pointer value that points to the object pointed to by p ”

- We have provided a formal semantics and a symbolic model checking procedure for the class of Linear Programs with the *undefined* symbol and with conditional expressions.
- We have proposed an alternative abstraction and refinement schema for a subset of the C programming language that employs linear arithmetics and arrays.
- Preliminary experimental results obtained with our prototype model checker EUREKA indicate that our procedure correctly analyses programs on which other tools either fail or provide incorrect answers.

Future Research Directions

- Find sufficient conditions for termination;
- Use of widening to enforce termination;
- Tighter integration with decision procedures;
- Application to pointers and dynamic arrays.

`www.ai.dist.unige.it/eureka`

Thank you!



A. Armando and C. Castellini and J. Mantovani,
Software Model Checking Using Linear Constraints,
Proc. of ICFEM, Seattle 2004, LNCS 3308, Springer.



Clark Barrett and Sergey Berezin,
CVC Lite: A new implementation of the Cooperating Validity Checker,
Proc. of CAV 2004, LNCS, Springer.



Leonardo de Moura and Harald Ruess and Natarajan Shankar and John Rushby,
The ICS decision procedure for embedded deduction,
Proc. of IJCAR 2004, LNCS 3097, Springer.



Thomas A. Henzinger and Ranjit Jhala and Rupak Majumdar and Gregoire Sutre,
Lazy abstraction,
Proc. of POPL 2002.



Cormac Flanagan,
Software Model Checking via Iterative Abstraction Refinement of Constraint Logic Queries,
CP+CV'04.