Program Analysis using SMT and MAX-SMT

Albert Rubio

joint work with

Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell

Universitat Politècnica de Catalunya

LOPSTR, September 2013

Outline

1. Introduction
2. SMT/Max-SMT solving
3. Invariant generation
4. Termination analysis
5. Further work

Outline

Motivation

- Develop static analysis tools
  - Fully automatic
  - Efficient
  - Scalable

Motivation

- Develop static analysis tools
  - Fully automatic
  - Efficient
  - Scalable
- Take advantatge of the new powerful arithmetic constraint solvers.

SMT-solvers

Constraint Based Program Analyisis techniques

Motivation

A particularly difficult verification problem:

- Prove termination of imperative programs automatically.
- Find ranking functions.
- Find supporting invariants.
- How to guide the search!.

Simple example

```
void simpleNT(int x, int y) {

  while (y>0) {

    while (x>0) {
      x=x-y;
      y=y-1;
    }
    y=y-1;
  }
}
```

Simple example

```
void simpleNT(int x, int y) {

  while (y>0) {

    while (x>0) {
      x=x-y;
      y=y-1;
    }
    y=y-1;
  }
}
```

Does not terminate. For instance, with x=3 and y=1

Simple example

```
void simpleT(int x, int y) {

  while (y>0) {

    while (x>0) {
      x=x-y;
      y=y+1;
    }
    y=y-1;
  }
}
```

Simple example

```
void simpleT(int x, int y) {

  while (y>0) {

    while (x>0) {
      x=x-y;
      y=y+1;
    }
    y=y-1;
  }
}
```

Terminates.

Simple example

```
void simpleT(int x, int y) {

  while (y>0) { Ranking function: y
    // Inv: y>0
    while (x>0) { Ranking function: x
      x=x-y;
      y=y+1;
    }
    y=y-1;
  }
}
```

Terminates.

Goals

- Present the constraint-based invariant generation method introduced by [Colón,Sankaranarayanan,Sipma 2003].
- Show how efficient SMT-solvers make it feasible in practice.
- Extend the method to generate Array invariants.
- Consider the termination problem within the constraint based method as in [Bradley,Manna,Sipma 2005].
- Show how to make it feasible in practice using Max-SMT

optimization instead of satisfaction

Outline

1. Introduction
2. SMT/Max-SMT solving
3. Invariant generation
4. Termination analysis
5. Further work

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ linear integer/real arithmetic.

$$(x < 0 \vee x \leq y \vee y < z) \wedge (x \geq 0) \wedge (x > y \vee y < z)$$

$$\{x = 1, y = 0, z = 2\}$$

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ linear integer/real arithmetic.

$$(x < 0 \vee x \leq y \vee \underline{y < z}) \wedge (\underline{x \geq 0}) \wedge (x > y \vee \underline{y < z})$$

$$\{x = 1, y = 0, z = 2\}$$

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ linear integer/real arithmetic.

$$(x < 0 \vee x \leq y \vee \underline{y < z}) \wedge (\underline{x \geq 0}) \wedge (x > y \vee \underline{y < z})$$

$$\{x = 1, y = 0, z = 2\}$$

There exist very efficient solvers: yices, z3, Barcelogic, ...
Can handle large formulas with a complex boolean structure.

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ non-linear (polynomial) integer/real arithmetic.

$$(x^2 + y^2 > 2 \vee x \cdot z \leq y \vee y \cdot z < z^2) \wedge (x > y \vee 0 < z)$$

$$\{x = 0, y = 1, z = 1\}$$

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ non-linear (polynomial) integer/real arithmetic.

$$(x^2 + y^2 > 2 \lor \underline{x \cdot z \leq y} \lor y \cdot z < z^2) \land (x > y \lor \underline{0 < z})$$

$$\{x = 0, y = 1, z = 1\}$$

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ non-linear (polynomial) integer/real arithmetic.

$$(x^2 + y^2 > 2 \vee \underline{x \cdot z \leq y} \vee y \cdot z < z^2) \wedge (x > y \vee \underline{0 < z})$$

$$\{x = 0, y = 1, z = 1\}$$

Non-linear arithmetic decidability:

- *Integers:* undecidable
- *Reals:* decidable **but** unpractical due to its complexity.

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: T = non-linear (polynomial) integer/real arithmetic.

$$(x^2 + y^2 > 2 \lor \underline{x \cdot z \leq y} \lor y \cdot z < z^2) \land (x > y \lor \underline{0 < z})$$

$$\{x = 0, y = 1, z = 1\}$$

Non-linear arithmetic decidability:

- *Integers:* undecidable
- *Reals:* decidable **but** unpractical due to its complexity.

Incomplete solvers focused on either satisfiability or unsatisfiability.

SMT solving

Input: Given a boolean formula φ over some theory T.

Question: Is there any interpretation that satisfies the formula?

Example: $T =$ non-linear (polynomial) integer/real arithmetic.

$$(x^2 + y^2 > 2 \vee \underline{x \cdot z \leq y} \vee y \cdot z < z^2) \wedge (x > y \vee \underline{0 < z})$$

$$\{x = 0, y = 1, z = 1\}$$

Non-linear arithmetic decidability:

- *Integers:* undecidable
- *Reals:* decidable **but** unpractical due to its complexity.

Incomplete solvers focused on either satisfiability or unsatisfiability.

Need to handle again large formulas with complex boolean structure.

Barcelogic SMT-solver works very well finding solutions

Optimization problems

(Weighted) Max-SMT problem

Input: Given an SMT formula $\varphi = C_1 \wedge \ldots \wedge C_m$ in CNF, where some of the clauses are *hard* and the others *soft* with a weight.

Output: An assignment for the hard clauses that minimizes the sum of the weights of the falsified soft clauses.

$$(x^2 + y^2 > 2 \vee x \cdot z \leq y \vee y \cdot z < z^2) \wedge (x > y \vee 0 < z \vee w(5)) \wedge \ldots$$

Outline

Invariants

Definition

An *invariant* of a program at a location is an assertion over the program variables that remains true whenever the location is reached.

Invariants

Definition

An *invariant* of a program at a location is an assertion over the program variables that remains true whenever the location is reached.

Definition

An invariant is said to be *inductive* at a program location if:

- *Initiation condition:* It holds the first time the location is reached.
- *Consecution condition:* It is preserved under every cycle back to the location.

Invariants

Definition

An *invariant* of a program at a location is an assertion over the program variables that remains true whenever the location is reached.

Definition

An invariant is said to be *inductive* at a program location if:

- *Initiation condition:* It holds the first time the location is reached.
- *Consecution condition:* It is preserved under every cycle back to the location.

We are focused on inductive invariants.

Constraint-based invariant generation

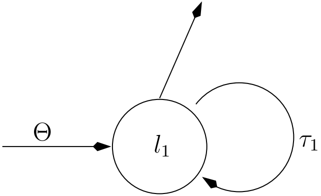
- Assume input programs consist of linear expressions
- Model the program as a *transition system*

Constraint-based invariant generation

- Assume input programs consist of linear expressions
- Model the program as a *transition system*

Simple example:

```
int main()
{
    int x;
    int y=-x;
l1: while (x>=0) {
      x--;
      y--;
    }
}
```



$\rho_\Theta : x' = x, \quad y' = -x$

$\rho_{\tau_1} : x \geq 0, \quad x' = x - 1, \quad y' = y - 1$

Constraint-based invariant generation

Assume we have a transition system with linear expressions.

Constraint-based invariant generation

Assume we have a transition system with linear expressions.

Keys:

Constraint-based invariant generation

Assume we have a transition system with linear expressions.

Keys:

- Use a template for candidate invariants.

$$c_1 x_1 + \ldots + c_n x_n + d \leq 0$$

Constraint-based invariant generation

Assume we have a transition system with linear expressions.

Keys:

- Use a template for candidate invariants.

$$c_1 x_1 + \ldots + c_n x_n + d \leq 0$$

- Check initiation and consecution conditions obtaining an $\exists\forall$ problem.

Constraint-based invariant generation

Assume we have a transition system with linear expressions.

Keys:

- Use a template for candidate invariants.

$$c_1x_1 + \ldots + c_nx_n + d \leq 0$$

- Check initiation and consecution conditions obtaining an $\exists\forall$ problem.
- Transform it using Farkas' Lemma into an $\exists$ problem over non-linear arithmetic.

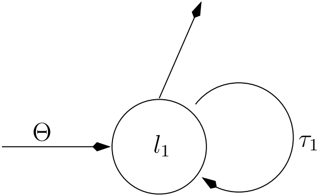
Constraint-based invariant generation

Following the example

Template invariant $I: \; c_1 x + c_2 y + d \leq 0$

Initiation: $\rho_\Theta \models I'$

Consecution: $\rho_{\tau_1} \wedge I \models I'$



$\rho_\Theta : x' = x, \quad y' = -x$
$\rho_{\tau_1} : x \geq 0, \quad x' = x - 1, \quad y' = y - 1$

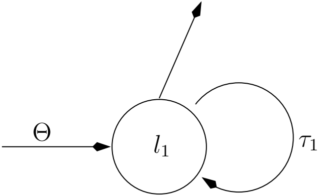
Constraint-based invariant generation

Following the example

Template invariant $I : c_1 x + c_2 y + d \leq 0$

$x' = x \wedge y' = -x \models c_1 x' + c_2 y' + d \leq 0$

Consecution: $\rho_{\tau_1} \wedge I \models I'$



$\rho_\Theta : x' = x, \quad y' = -x$
$\rho_{\tau_1} : x \geq 0, \quad x' = x - 1, \quad y' = y - 1$

Constraint-based invariant generation

We need to solve: $\exists c_1, c_2, d\, \forall x, y, x', y'$

Initiation:

$$x' = x \wedge y' = -x \;\models\; c_1 x' + c_2 y' + d \leq 0$$

Consecution:

$$x \geq 0 \wedge x' = x - 1 \wedge y' = y - 1 \wedge c_1 x + c_2 y + d \leq 0 \models c_1 x' + c_2 y' + d \leq 0$$

Use Farkas' Lemma to remove the universal quantifiers

Farkas' Lemma

Farkas' Lemma:

$$(\forall \overline{x}) \begin{bmatrix} a_{11}x_1 + \cdots + & a_{1n}x_n + & b_1 \leq 0 \\ \vdots & \vdots & \vdots \leq 0 \\ a_{m1}x_1 + \cdots + & a_{mn}x_n + & b_m \leq 0 \end{bmatrix} \Rightarrow \varphi : e_1 x_1 + \ldots + e_n x_n + e_0 \leq 0$$

$$\Leftrightarrow$$

$$\exists \lambda_0, \lambda_1, \ldots, \lambda_m \geq 0,$$

$$e_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \ \ldots \ , e_n = \sum_{i=1}^{m} \lambda_i a_{in}, \ e_0 = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$$

or

$$0 = \sum_{i=1}^{m} \lambda_i a_{i1}, \ \ldots \ , 0 = \sum_{i=1}^{m} \lambda_i a_{in}, \ 1 = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$$

Farkas' Lemma

Farkas' Lemma:

$$(\forall \overline{x}) \begin{bmatrix} a_{11}x_1 + \cdots + & a_{1n}x_n + & b_1 \leq 0 \\ \vdots & \vdots & \vdots \leq 0 \\ a_{m1}x_1 + \cdots + & a_{mn}x_n + & b_m \leq 0 \end{bmatrix} \Rightarrow \varphi : e_1x_1 + \ldots + e_nx_n + e_0 \leq 0$$

$$\Leftrightarrow \ \exists \lambda_0, \lambda_1, \ldots, \lambda_m \geq 0,$$

$$\begin{array}{ccccccccccc}
\lambda_1 & * & a_{11} & x_1 & + & \cdots & + & a_{1n} & x_n & + & b_1 & \leq 0 \\
 & & \vdots & & & & & \vdots & & & \vdots & \leq 0 \\
\lambda_m & * & a_{m1} & x_1 & + & \cdots & + & a_{mn} & x_n & + & b_m & \leq 0 \\
\hline
 & & e_1 & x_1 & + & \cdots & + & e_n & x_n & + & d & \leq 0 \\
\textit{or} & & & & & & & & & & & \\
 & & 0 & & + & \cdots & + & 0 & & + & 1 & \leq 0
\end{array}$$

Farkas' Lemma

Farkas' Lemma:

$$(\forall \overline{x}) \begin{bmatrix} a_{11}x_1 + \cdots + a_{1n}x_n + b_1 \le 0 \\ \vdots \qquad\qquad \vdots \qquad \vdots \le 0 \\ a_{m1}x_1 + \cdots + a_{mn}x_n + b_m \le 0 \end{bmatrix} \Rightarrow \varphi : e_1x_1 + \ldots + e_nx_n + e_0 \le 0$$

$$\Leftrightarrow \ \exists \lambda_0, \lambda_1, \ldots, \lambda_m \ge 0,$$

$$\begin{array}{llllllllll}
\lambda_0 & * & & & & & & & -1 & \le 0 \\
\lambda_1 & * & a_{11} & x_1 & + \cdots + & a_{1n} & x_n & + & b_1 & \le 0 \\
 & & \vdots & & & \vdots & & & \vdots & \le 0 \\
\lambda_m & * & a_{m1} & x_1 & + \cdots + & a_{mn} & x_n & + & b_m & \le 0 \\
\hline
 & & e_1 & x_1 & + \cdots + & e_n & x_n & + & d & \le 0 \\
\textit{or} & & & & & & & & & \\
 & & 0 & & + \cdots + & 0 & & + & 1 & \le 0
\end{array}$$

Farkas' Lemma

Farkas' Lemma:

$$(\forall \overline{x}) \begin{bmatrix} a_{11}x_1 + \cdots + & a_{1n}x_n + & b_1 \leq 0 \\ \vdots & \vdots & \vdots \leq 0 \\ a_{m1}x_1 + \cdots + & a_{mn}x_n + & b_m \leq 0 \end{bmatrix} \Rightarrow \varphi : e_1x_1 + \ldots + e_nx_n + e_0 \leq 0$$

$$\Leftrightarrow \ \exists \lambda_0, \lambda_1, \ldots, \lambda_m \geq 0,$$

$$\begin{array}{ll|cccc} & & x_1 & \cdots & x_n & \\ \hline \lambda_0 & * & & & & -1 \\ \lambda_1 & * & a_{11} & \cdots & a_{1n} & b_1 \\ & & \vdots & & \vdots & \vdots \\ \lambda_m & * & a_{m1} & \cdots & a_{mn} & b_m \\ \hline & & e_1 & \cdots & e_n & d \\ \textit{or} & & & & & \\ & & 0 & \cdots & 0 & 1 \end{array}$$

Farkas' Lemma

Farkas' Lemma:

$$(\forall \overline{x}) \begin{bmatrix} a_{11}x_1 + \cdots + & a_{1n}x_n + & b_1 \leq 0 \\ \vdots & \vdots & \vdots \leq 0 \\ a_{m1}x_1 + \cdots + & a_{mn}x_n + & b_m \leq 0 \end{bmatrix} \Rightarrow \varphi : e_1 x_1 + \ldots + e_n x_n + e_0 \leq 0$$

$$\Leftrightarrow \ \exists \lambda_0, \lambda_1, \ldots, \lambda_m \geq 0,$$

$$e_1 = \sum_{i=1}^{m} \lambda_i a_{i1}, \ \ldots \ , e_n = \sum_{i=1}^{m} \lambda_i a_{in}, \ e_0 = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$$

or

$$0 = \sum_{i=1}^{m} \lambda_i a_{i1}, \ \ldots \ , 0 = \sum_{i=1}^{m} \lambda_i a_{in}, \ 1 = (\sum_{i=1}^{m} \lambda_i b_i) - \lambda_0$$

Farkas' Lemma

Farkas' Lemma: our example

Initiation condition: $x' = x \wedge y' = -x \models c_1x' + c_2y' + d \leq 0$

$$(\forall x, y, x', y') \begin{bmatrix} -1x + 0y + 1x' + 0y' + 0 \leq 0 \\ 1x + 0y + -1x' + 0y' + 0 \leq 0 \\ \\ \end{bmatrix} \Rightarrow 0x + 0y + c_1x' + c_2y' + d \leq 0$$

$$\Leftrightarrow$$

$$\exists \lambda_0^i \geq 0, \lambda_1^i \geq 0, \lambda_2^i \geq 0, \ldots$$

Farkas' Lemma

Farkas' Lemma: our example

Initiation condition: $x' - x = 0 \wedge y' + x = 0 \models c_1x' + c_2y' + d \leq 0$

$$(\forall x, y, x', y')\left[-1x + 0y + 1x' + 0y' + 0 = 0\right] \Rightarrow 0x + 0y + c_1x' + c_2y' + d \leq 0$$

$$\Leftrightarrow$$

$$\exists \lambda_0^i \geq 0, \lambda_1^i, \ldots$$

Farkas' Lemma

Farkas' Lemma: our example

Initiation condition: $x' - x = 0 \wedge y' + x = 0 \models c_1 x' + c_2 y' + d \leq 0$

$$(\forall x, y, x', y') \begin{bmatrix} -1x + 0y + 1x' + 0y' + 0 = 0 \\ 1x + 0y + 0x' + 1y' + 0 = 0 \end{bmatrix} \Rightarrow 0x + 0y + c_1 x' + c_2 y' + d \leq 0$$

$$\Leftrightarrow$$

$$\exists \lambda_0^i \geq 0, \lambda_1^i, \lambda_2^i$$

Farkas' Lemma

Farkas' Lemma: our example

Initiation condition: $x' - x = 0 \wedge y' + x = 0 \models c_1 x' + c_2 y' + d \leq 0$

$$
\begin{array}{cc|ccccc}
 & & x & y & x' & y' & \\
\hline
\lambda_0^i & * & & & & & -1 \\
\lambda_1^i & * & -1 & 0 & 1 & 0 & 0 \\
\lambda_2^i & * & 1 & 0 & 0 & 1 & 0 \\
\hline
 & & 0 & 0 & c_1 & c_2 & d \\
\textit{or} & & & & & & \\
 & & 0 & 0 & 0 & 0 & 1
\end{array}
$$

$$\Leftrightarrow$$

$$\exists \lambda_0^i \geq 0, \lambda_1^i, \lambda_2^i$$

Farkas' Lemma

Farkas' Lemma: our example

Initiation condition: $x' - x = 0 \wedge y' + x = 0 \models c_1x' + c_2y' + d \leq 0$

$$
\begin{array}{cc|rrrr|r}
 & & x & y & x' & y' & \\
\hline
\lambda_0^i & * & & & & & -1 \\
\lambda_1^i & * & -1 & 0 & 1 & 0 & 0 \\
\lambda_2^i & * & 1 & 0 & 0 & 1 & 0 \\
\hline
 & & 0 & 0 & c_1 & c_2 & d \\
\textit{or} & & & & & & \\
 & & 0 & 0 & 0 & 0 & 1
\end{array}
$$

$$\Leftrightarrow$$

$$\exists \lambda_0^i \geq 0, \lambda_1^i, \lambda_2^i, c_1, c_2, d$$

$$0 = -\lambda_1^i + \lambda_2^i, \;\; c_1 = \lambda_1^i, \;\; c_2 = \lambda_2^i, \;\; d = -\lambda_0^i$$

or

$$0 = -\lambda_1^i + \lambda_2^i, \;\; 0 = \lambda_1^i, \;\; 0 = \lambda_2^i, \;\; 1 = -\lambda_0^i$$

Farkas' Lemma

Farkas' Lemma: our example
Consecution condition:

$$x \geq 0 \wedge x' = x - 1 \wedge y' = y - 1 \wedge c_1 x + c_2 y + d \leq 0 \models c_1 x' + c_2 y' + d \leq 0$$

Farkas' Lemma

Farkas' Lemma: our example
Consecution condition:

$$-x \leq 0 \wedge x' - x + 1 = 0 \wedge y' - y + 1 = 0 \wedge c_1 x + c_2 y + d \leq 0 \models c_1 x' + c_2 y' + d \leq 0$$

Farkas' Lemma

Farkas' Lemma: our example
Consecution condition:

$$-x \leq 0 \wedge x' - x + 1 = 0 \wedge y' - y + 1 = 0 \wedge c_1 x + c_2 y + d \leq 0 \models c_1 x' + c_2 y' + d \leq 0$$

$$
\begin{array}{cc|ccccc}
 & & x & y & x' & y' & \\
\hline
\lambda_0^c & * & & & & & -1 \\
\lambda_1^c & * & -1 & 0 & 0 & 0 & 0 \\
\lambda_2^c & * & -1 & 0 & 1 & 0 & 1 \\
\lambda_3^c & * & 0 & -1 & 0 & 1 & 1 \\
\lambda_4^c & * & c_1 & c_2 & 0 & 0 & d \\
\hline
 & & 0 & 0 & c_1 & c_2 & d \\
or & & & & & & \\
 & & 0 & 0 & 0 & 0 & 1
\end{array}
$$

Farkas' Lemma

Farkas' Lemma: our example
Consecution condition:

$$-x \leq 0 \wedge x' - x + 1 = 0 \wedge y' - y + 1 = 0 \wedge c_1 x + c_2 y + d \leq 0 \models c_1 x' + c_2 y' + d \leq 0$$

$$\exists \lambda_0^c \geq 0, \lambda_1^c \geq 0, \lambda_2^c, \lambda_3^c, \lambda_4^c \geq 0, c_1, c_2, d$$

$$0 = -\lambda_1^c - \lambda_2^c + \lambda_4^c c_1, \;\; 0 = -\lambda_3^c + \lambda_4^c c_2, \;\; c_1 = \lambda_2^c, \;\; c_2 = \lambda_3^c, \;\; d = -\lambda_0^c + \lambda_2^c + \lambda_3^c + \lambda_4^c d$$

or

$$0 = -\lambda_1^c - \lambda_2^c + \lambda_4^c c_1, \;\; 0 = -\lambda_3^c + \lambda_4^c c_2, \;\; 0 = \lambda_2^c, \;\; 0 = \lambda_3^c, \;\; 1 = -\lambda_0^c + \lambda_2^c + \lambda_3^c + \lambda_4^c d$$

Farkas' Lemma

Farkas' Lemma: our example

$$\exists \lambda_0^i \geq 0, \lambda_1^i, \lambda_2^i, \lambda_0^c \geq 0, \lambda_1^c \geq 0, \lambda_2^c, \lambda_3^c, \lambda_4^c \geq 0, c_1, c_2, d$$

$$(0 = -\lambda_1^i + \lambda_2^i, \;\; c_1 = \lambda_1^i, \;\; c_2 = \lambda_2^i, \;\; d = -\lambda_0^i$$

or

$$0 = -\lambda_1^i + \lambda_2^i, \;\; 0 = \lambda_1^i, \;\; 0 = \lambda_2^i, \;\; 1 = -\lambda_0^i)$$

and

$$(0 = -\lambda_1^c - \lambda_2^c + \lambda_4^c c_1, \;\; 0 = -\lambda_3^c + \lambda_4^c c_2, \;\; c_1 = \lambda_2^c, \;\; c_2 = \lambda_3^c, \;\; d = -\lambda_0^c + \lambda_2^c + \lambda_3^c + \lambda_4^c d$$

or

$$0 = -\lambda_1^c - \lambda_2^c + \lambda_4^c c_1, \;\; 0 = -\lambda_3^c + \lambda_4^c c_2, \;\; 0 = \lambda_2^c, \;\; 0 = \lambda_3^c, \;\; 1 = -\lambda_0^c + \lambda_2^c + \lambda_3^c + \lambda_4^c d)$$

Solution: $c_1 = 1, \; c_2 = 1, \; d = 0$. Hence $x + y \leq 0$ is invariant.

Invariant generation process

- Input: A C++ program
- Output: A set of independent invariants for some locations

Basic procedure:

- Template invariant: $c_1x + c_2y + d \leq 0$
- Send the non-linear formula to Barcelogic
- Add the obtained invariant to the transition system
- Iterate or quit if no new invariant is obtained

Invariant generation process

An Incremental algorithm producing non-redundant invariants:

- Let Inv be the set of already generated invariants.
- To avoid generation of redundant invariants add

$$\exists x \exists y (Inv \ \wedge \ c_1 x + c_2 y + d > 0)$$

Note that

- it is also existentially quantified
- it is also nonlinear arithmetic

Invariant generation process

- Input: A C++ program
- Output: A set of independent invariants for some locations

Basic procedure:

- Template invariant: $c_1 x + c_2 y + d \leq 0$
- Send the non-linear formula to Barcelogic
- Add the obtained invariant to the transition system
- Iterate or quit if no new invariant is obtained

This is what we do!

Invariant generation with arrays

Goal:

Invariant generation with arrays

Goal:

- Discovering invariant properties on values of array elements and other program variables.

Invariant generation with arrays

Goal:

- Discovering invariant properties on values of array elements and other program variables.
- Focused on *universally* quantified array invariants.

Invariant generation with arrays

Goal:

- Discovering invariant properties on values of array elements and other program variables.
- Focused on *universally* quantified array invariants.
- Using an automatic generation process.

Invariant generation with arrays

Goal:

- Discovering invariant properties on values of array elements and other program variables.
- Focused on *universally* quantified array invariants.
- Using an automatic generation process.

However, most of the existing techniques need some guidance.

Examples

Palindrome array:

```
int main() {
  const int N;
  assume(N >= 0);
  int A[N];
  int i = 0;
  while (i < N/2) {
    if (A[i] != A[N-i-1])
      break;
    ++i;
  }
}
```

$$\forall\alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = A[N-\alpha-1]$$

Examples

Array initialization:

```
int main() {
  const int N;
  assume(N >= 0);
  int A[N];
  int i = 0;
  while (i < N) {
    A[i] = 2i+N-1;
    i++;
  }
}
```

$\forall\alpha : \ 0 \leq \alpha \leq i-1 : \ A[\alpha] = 2\alpha + N - 1$

Array invariant language

Programs are assumed to consist of *unnested* loops and linear assignments, conditions and array accesses.

Array invariant language

Programs are assumed to consist of *unnested* loops and linear assignments, conditions and array accesses.

To simplify assume we have a single occurrence of an array variable.

Array invariant language

Programs are assumed to consist of *unnested* loops and linear assignments, conditions and array accesses.

To simplify assume we have a single occurrence of an array variable.

Our method generates invariants of the form:

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

where $\mathcal{C}$, $\mathcal{E}$ and $\mathcal{B}$ are linear expressions with integer coefficients over the scalar variables of the program $\overline{v} = (v_1, \ldots, v_n)$ and $a, d, b_\alpha \in \mathbb{Z}$.

Array invariant language

Programs are assumed to consist of *unnested* loops and linear assignments, conditions and array accesses.

To simplify assume we have a single occurrence of an array variable.

Our method generates invariants of the form:

$$\forall\alpha \ : \ 0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \le \ 0$$

where $\mathcal{C}$, $\mathcal{E}$ and $\mathcal{B}$ are linear expressions with integer coefficients over the scalar variables of the program $\overline{v} = (v_1, \ldots, v_n)$ and $a, d, b_\alpha \in \mathbb{Z}$.

Easily extensible to m array variables and k occurrences:

$$\forall\alpha \ : \ 0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \ : \ \Sigma_{i=1}^{m}\Sigma_{j=1}^{k} a_{ij} A_i[d_{ij}\alpha + \mathcal{E}_{ij}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \le \ 0$$

Examples

Palindrome array:

```
int main() {
  const int N;
  assume(N >= 0);
  int A[N];
  int i = 0;
  while (i < N/2) {
    if (A[i] != A[N-i-1])
      break;
    ++i;
  }
}
```

$$\forall \alpha : \ 0 \leq \alpha \leq i - 1 : \ A[\alpha] = A[N - \alpha - 1]$$

Examples

Palindrome array:

```
int main() {
  const int N;
  assume(N >= 0);
  int A[N];
  int i = 0;
  while (i < N/2) {
    if (A[i] != A[N-i-1])
      break;
    ++i;
  }
}
```

$\forall\alpha : \ 0 \le \alpha \le i-1 : \ A[\alpha] - A[N-\alpha-1] \le 0$
$\forall\alpha : \ 0 \le \alpha \le i-1 : \ A[N-\alpha-1] - A[\alpha] \le 0$

Existing approaches for array invariant generation

Abstract interpretation [Gopan,Reps,Sagiv 2005; Halbwachs,Peron 2008]

Predicate abstraction [Flanagan,Qadeer 2002; Lahiri,Bryant 2004; Jhala,McMillan 2007; Srivastava,Gulwani 2009]

First-order theorem proving [Kovács,Voronkov 2009; McMillan 2008]

Computational algebra [Henzinger,Hottelier,Kovács,Rybalchenko 2010]

Existing approaches for array invariant generation

Abstract interpretation [Gopan,Reps,Sagiv 2005; Halbwachs,Peron 2008]

Predicate abstraction [Flanagan,Qadeer 2002; Lahiri,Bryant 2004; Jhala,McMillan 2007; Srivastava,Gulwani 2009]

First-order theorem proving [Kovács,Voronkov 2009; McMillan 2008]

Computational algebra [Henzinger,Hottelier,Kovács,Rybalchenko 2010]

Constraint-based invariant generation [Larraz,Rodríguez,Rubio 2013]

Ideas behind the method

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

Ideas behind the method: 3 phases

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_{\alpha} \cdot \alpha \ \leq \ 0$$

Ideas behind the method: 3 phases

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

Ideas behind the method: 3 phases

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

Ideas behind the method: 3 phases

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

Ideas behind the method: Phase 1

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \le \ 0$$

$$\mathcal{C}(\overline{v}) = c_1 v_1 + \ldots + c_n v_n + c_{n+1}$$

Ideas behind the method: Phase 1

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall \alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

$$\mathcal{C}(\overline{v}) = c_1 v_1 + \ldots + c_n v_n + c_{n+1}$$

Initiation condition: the first time the location is reached it holds that $\mathcal{C}(\overline{v'}) = 0$, i.e., the domain is empty.

Ideas behind the method: Phase 1

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \; : \; 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \; : \; a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \; + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \; \leq \; 0$$

$$\mathcal{C}(\overline{v}) = c_1 v_1 + \ldots + c_n v_n + c_{n+1}$$

Initiation condition: the first time the location is reached it holds that $\mathcal{C}(\overline{v'}) = 0$, i.e., the domain is empty.

Consecution condition: after every cycle back to the location it holds that either $\mathcal{C}(\overline{v'}) = \mathcal{C}(\overline{v})$ or $\mathcal{C}(\overline{v'}) = \mathcal{C}(\overline{v}) + 1$

Ideas behind the method: Phase 2

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \;:\; 0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \;:\; a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \; + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \;\le\; 0$$

$$d,\; \mathcal{E}(\overline{v}) = e_1 v_1 + \ldots + e_n v_n + e_{n+1}$$

Ideas behind the method: Phase 2

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \le \ 0$$

$$d, \ \mathcal{E}(\overline{v}) = e_1 v_1 + \ldots + e_n v_n + e_{n+1}$$

Indexes are valid: $0 \le \alpha \le \mathcal{C}(\overline{v'}) - 1 \implies 0 \le d\alpha + \mathcal{E}(\overline{v'}) \le |A| - 1$

Ideas behind the method: Phase 2

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

$$d, \ \mathcal{E}(\overline{v}) = e_1 v_1 + \ldots + e_n v_n + e_{n+1}$$

Indexes are valid: $0 \leq \alpha \leq \mathcal{C}(\overline{v'}) - 1 \implies 0 \leq d\alpha + \mathcal{E}(\overline{v'}) \leq |A| - 1$

No array update index is in $\{d \cdot \alpha + \mathcal{E}(\overline{v}) \mid 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1\}$, i.e., elements for which invariant held in previous iterations are not modified.

Ideas behind the method: Phase 3

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

$$a, \ b_\alpha, \ \mathcal{B}(\overline{v}) = b_1 v_1 + \ldots + b_n v_n + b_{n+1}$$

Ideas behind the method: Phase 3

Find conditions ensuring inductive invariance and represent them as implications of templates.

$$\forall\alpha \ : \ 0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \le \ 0$$

$$a, \ b_\alpha, \ \mathcal{B}(\overline{v}) = b_1 v_1 + \ldots + b_n v_n + b_{n+1}$$

The property keeps holding for unchanged array elements:

$$0 \le \alpha \le \mathcal{C}(\overline{v}) - 1 \ \wedge \ x + \mathcal{B}(\overline{v}) + b_\alpha \alpha \ \le \ 0 \Rightarrow x + \mathcal{B}(\overline{v'}) + b_\alpha \alpha \ \le \ 0$$

The property holds for some new consecutive array element:

$$a \cdot A[d \cdot \mathcal{C}(\overline{v}) + \mathcal{E}(\overline{v'})] \ + \mathcal{B}(\overline{v'}) + b_\alpha \cdot \mathcal{C}(\overline{v}) \ \le \ 0$$

Ideas behind the method: Result

As a result, every solution found after the three phases provides an array invariant of the form:

$$\forall\alpha \ : \ 0 \leq \alpha \leq \mathcal{C}(\overline{v}) - 1 \ : \ a \cdot A[d \cdot \alpha + \mathcal{E}(\overline{v})] \ + \mathcal{B}(\overline{v}) + b_\alpha \cdot \alpha \ \leq \ 0$$

where $\mathcal{C}$, $\mathcal{E}$ and $\mathcal{B}$ are linear polynomials with integer coefficients over the scalar variables of the program $\overline{v} = (v_1, \ldots, v_n)$ and $a, d, b_\alpha \in \mathbb{Z}$.

Examples

Palindrome array:

```
int main() {
  const int N;
  assume(N >= 0);
  int A[N];
  int i = 0;
  while (i < N/2) {
    if (A[i] != A[N-i-1])
      break;
    ++i;
  }
}
```

$$\forall\alpha:\ 0 \le \alpha \le i-1:\ A[\alpha] - A[N-\alpha-1] \le 0$$
$$\forall\alpha:\ 0 \le \alpha \le i-1:\ A[N-\alpha-1] - A[\alpha] \le 0$$

Examples

Array initialization:

```
int main() {
  const int N;
  assume(N >= 0);
  int A[N];
  int i = 0;
  while (i < N) {
    A[i] = 2i+N-1;
    i++;
  }
}
```

$\forall\alpha :\ 0 \le \alpha \le i-1 :\ A[\alpha] - 2\alpha - N + 1 \le 0$

$\forall\alpha :\ 0 \le \alpha \le i-1 :\ -A[\alpha] + 2\alpha + N - 1 \le 0$

Other examples we can handle

```
int main() {  // Heap property
  const int N;
  assume(N >= 0);
  int A[2*N], i;
  i=0;
  while (2*i+2 < 2*N) {
    if (A[i]>A[2*i+1] or A[i]>A[2*i+2])
      break;
    ++i;
  }
}
```

$\forall \alpha : \; 0 \leq \alpha \leq i-1 : \; A[\alpha] \leq A[2\alpha+2]$ $\forall \alpha : \; 0 \leq \alpha \leq i-1 : \; A[\alpha] \leq A[2\alpha+1]$

Other examples we can handle

```
int main() {  // Partial initialization [GopanRepsSavig05]
  const int N;
  assume(N >= 0);
  int A[N], B[N], C[N];
  int i=0, j=0;
  while (i < N) {
    if (A[i] == B[i])
      C[j++] = i;
    ++i;
  }
}
```

$\forall\alpha :\ 0 \le \alpha \le j-1 :\ C[\alpha] \le \alpha + i - j$

$\forall\alpha :\ 0 \le \alpha \le j-1 :\ C[\alpha] \ge \alpha$

Other examples we can handle

```
int main() {  // Array insertion
  const int N;
  int A[N], i, j, x;
  assume(0 <= i and i < N);
  x = A[i];
  j = i-1;
  while (j >= 0 and A[j] > x) {
    A[j+1] = A[j];
    --j;
  }
}
```

$$\forall \alpha : \; 0 \leq \alpha \leq i - j - 2 : \; A[i - \alpha] \geq x + 1$$

Extensions: Weakening the condition on the initial domain

We can try to extend the empty universally quantified domain of α.

```
int main() {  // Array maximum
  const int N;
  assume(N > 0);
  int A[N], i=1;
  int max = A[0];
  while (i<N) {
    if (max<A[i]) max=A[i];
    ++i;
  }
}
```

$$\forall \alpha : \ 0 \le \alpha \le i - 2 : \ A[\alpha + 1] \le max$$

Extensions: Weakening the condition on the initial domain

We can try to extend the empty universally quantified domain of α.

```
int main() {  // Array maximum
  const int N;
  assume(N > 0);
  int A[N], i=1;
  int max = A[0];
  while (i<N) {
    if (max<A[i]) max=A[i];
    ++i;
  }
}
```

$\forall\alpha : \ 0 \leq \alpha \leq i-2 : \ A[\alpha+1] \leq max$

$\forall\alpha : \ 1 \leq \alpha \leq i-1 : \ A[\alpha] \leq max$

Extensions: Weakening the condition on the initial domain

We can try to extend the empty universally quantified domain of α.

```
int main() {  // Array maximum
  const int N;
  assume(N > 0);
  int A[N], i=1;
  int max = A[0];
  while (i<N) {
    if (max<A[i]) max=A[i];
    ++i;
  }
}
```

$\forall\alpha:\ 0 \leq \alpha \leq i-2:\ A[\alpha+1] \leq max$

$\forall\alpha:\ 0 \leq \alpha \leq i-1:\ A[\alpha] \leq max$ (*extended*)

Extensions: Relaxation of the increment step

We can allow $\mathcal{C}(\overline{v})$ to increase more than one by one.

```
int main() {  // Array minimum and maximum
  int A[2*N], i;
  int min = A[0];
  int max = A[0];
  for (i = 1; i+1 < N; i += 2) {
    int tmpmin, tmpmax;
    if (A[i] < A[i+1]) { tmpmin = A[ i ]; tmpmax = A[i+1]; }
    else { tmpmin = A[i+1]; tmpmax = A[ i ]; }
    if (max < tmpmax) max = tmpmax;
    if (min > tmpmin) min = tmpmin;
  }
}
```

$\forall\alpha:\ 0 \le \alpha \le i-1:\ A[\alpha] \ge min \ \wedge\ A[\alpha] \le max$

Extensions: Addition of element order assumptions

We can take into account that an array is *sorted*.

```
int main() {  // First occurrence
  const int N;
  assume(N >= 0);
  int A[N], x = getX();
  int l=0, u=N;
  // Pre: A is sorted in ascending order
  while (l < u) {
    int m = (l+u)/2;
    if (A[m]<x) l=m+1; else u=m;
  }
}
```

$\forall \alpha : \ 0 \leq \alpha \leq l-1 : \ A[\alpha] < x$

$\forall \alpha : \ 0 \leq \alpha \leq N-1-u : \ A[N-1-\alpha] \leq x$

Experiments with (real) code

Our techniques have been implemented in a tool called `cppinv`.

As a challenging set of benchmarks we have used code made by undergraduate students for solving the *first occurrence* problem in a sorted array (taken from a programming learning environment Jutge.org)

Experiments with (real) code

Our techniques have been implemented in a tool called `cppinv`.

As a challenging set of benchmarks we have used code made by undergraduate students for solving the *first occurrence* problem in a sorted array (taken from a programming learning environment Jutge.org)

In contrast to the standard academic examples the code is:

- involved and ugly
- unnecessary conditional statements
- includes repeated code

Experiments with (real) code

Our techniques have been implemented in a tool called `cppinv`.

As a challenging set of benchmarks we have used code made by undergraduate students for solving the *first occurrence* problem in a sorted array (taken from a programming learning environment Jutge.org)

In contrast to the standard academic examples the code is:

- involved and ugly
- unnecessary conditional statements
- includes repeated code

All *nice* properties we need for testing our tool!

Examples of students' code

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);

  int e = 0, d = N - 1, m, pos;
  bool found = false, exit  = false;
  while (e <= d and not exit) {
    m = (e+d)/2;
    if (x > A[m]) {
      if (not found) e = m+1;
      else exit = true;
    } else if (x < A[m]) {
        if (not found) d = m-1;
        else exit = true;
      } else {
          found = true; pos = m; d = m-1;
        }
  }

  if (found) {
    while (x  == A[pos-1]) --pos;
    return pos; }
  return -1;
}
```

```
int first_occurrence(int x, int A[N]) {
  assume(N > 0);

  int l=0, u=N;

  while (l < u) {
    int m = (l+u)/2;
    if (A[m]<x) l=m+1;
    else u=m;
  }

 if (l>=N || A[l]!=x) l=-1;
 return l;
}
```

Examples of students' code

- We have checked the 38 accepted (as correct) iterative instances.

Examples of students' code

- We have checked the 38 accepted (as correct) iterative instances.
- Our tool was always able to find both standard inavariants.
- The time consumed was very different depending on how involved the code was.

Examples of students' code

- We have checked the 38 accepted (as correct) iterative instances.
- Our tool was always able to find both standard inavariants.
- The time consumed was very different depending on how involved the code was.
- The main efficiency problem of our tool is that it is exhaustive.

Outline

Motivation:

- Prove termination of imperative programs automatically.
- Find ranking functions.
- Find supporting invariants.
- How to guide the search!.

Ranking functions and Invariants

Basic method: find a single *ranking function* f : States $\rightarrow \mathbb{Z}$, with $f(S) \geq 0$ and $f(S) > f(S')$ after every iteration.

Ranking functions and Invariants

Basic method: find a single *ranking function* f : States $\rightarrow \mathbb{Z}$, with $f(S) \geq 0$ and $f(S) > f(S')$ after every iteration.
It does not work in practice in many cases.
What is (at least) necessary?

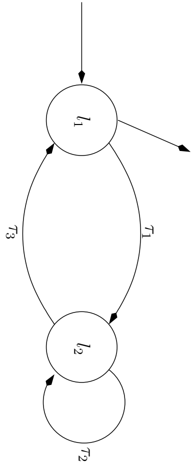
Ranking functions and Invariants

Basic method: find a single *ranking function* f : States $\rightarrow \mathbb{Z}$, with $f(S) \geq 0$ and $f(S) > f(S')$ after every iteration.
It does not work in practice in many cases.
What is (at least) necessary?

- Find supporting Invariants
- Consider a (lexicographic) combination of ranking functions

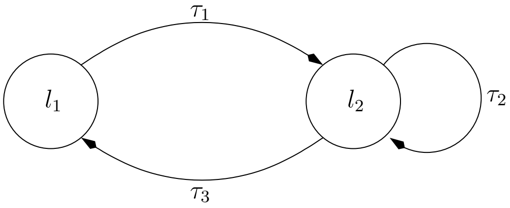
Ranking functions and Invariants: Example

```
int main()
{
    int x=indet(),y=indet(),z=indet();
l1: while (y>=1) {
      x--;
l2:   while (y<z) {
        x++; z--;
      }
      y=x+y;
    }
}
```



Ranking functions and Invariants: Example

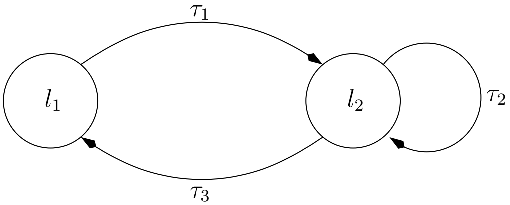
Transition system:



$$
\begin{array}{llllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

Ranking functions and Invariants: Example

Transition system:

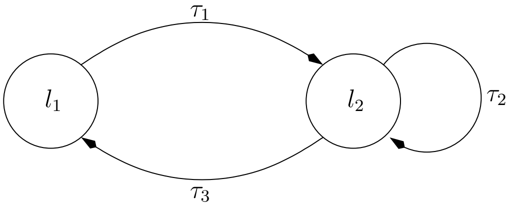


$$
\begin{array}{llllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

$f(x, y, z) = z$ is a ranking function for τ_2

Ranking functions and Invariants: Example

Transition system:

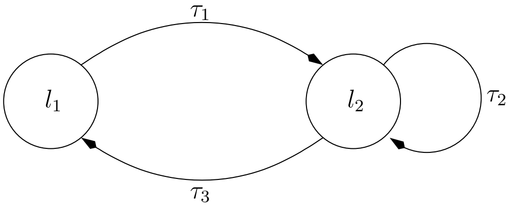


$$\begin{array}{llllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}$$

It is necessary a supporting invariant $y \geq 1$ at ℓ_2.

Ranking functions and Invariants: Example

Transition system:

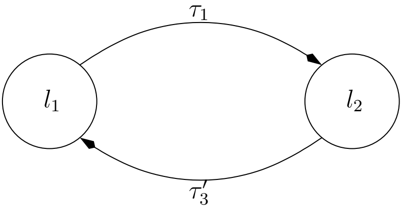


$$
\begin{array}{lllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

We can discard all executions that pass through τ_2.

Ranking functions and Invariants: Example

Transition system:



$$\rho_{\tau_1}: \quad y \geq 1, \quad x' = x - 1, \quad y' = y, \quad z' = z$$
$$\rho_{\tau_3'}: \quad y \geq 1, \quad y \geq z, \quad x' = x, \quad y' = x + y, \quad z' = z$$

We can discard all executions that pass through τ_2.

Ranking functions and Invariants

In order to discard a transition τ_i we need to find a ranking function f over the integers such that:

1. $\tau_i \Longrightarrow f(x_1, \ldots, x_n) \geq 0$ (bounded)
2. $\tau_i \Longrightarrow f(x_1, \ldots, x_n) > f(x'_1, \ldots, x'_n)$ (strict-decreasing)
3. $\tau_j \Longrightarrow f(x_1, \ldots, x_n) \geq f(x'_1, \ldots, x'_n)$ for all j (non-increasing)

Ranking functions and Invariants: Combined problem

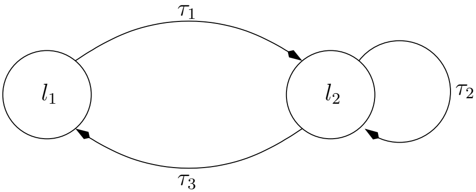
In order to prove properties of the ranking function we may need to generate invariants.

Generation of both invariants and ranking functions should be combined in the same satisfaction problem.

Both are found at the same time [BMS2005].

Ranking functions and Invariants: Example

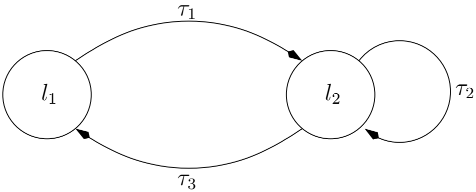
Transition system:



$$\begin{array}{llllll}
\rho_{\tau_1}: & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}$$

Ranking functions and Invariants: Example

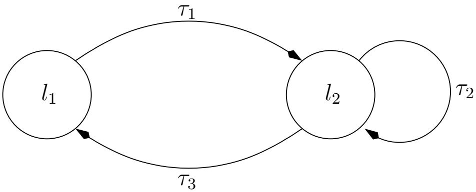
Transition system:



$$
\begin{array}{llllll}
\rho_{\tau_1}: & {\color{blue}l_1}, & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & {\color{blue}l_2}, & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & {\color{blue}l_2}, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

Ranking functions and Invariants: Example

Transition system:

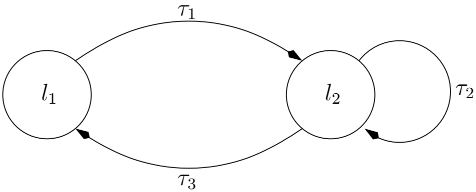


$$
\begin{array}{llllll}
\rho_{\tau_1'}: & 0 \leq 0, & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y \geq 1, & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

and ranking function $f(x, y, z) = z$, fulfiling all properties for τ_2

Ranking functions and Invariants: Example

Transition system:

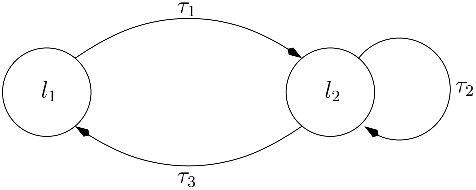


$$\begin{array}{llllll}
\rho_{\tau_1}: & & y \geq 1, & x' = x-1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y \geq 1, & y < z, & x' = x+1, & y' = y, & z' = z-1 \\
\rho_{\tau_3}: & y \geq 1, & y \geq z, & x' = x, & y' = x+y, & z' = z
\end{array}$$

and ranking function $f(x,y,z) = z$, fulfiling all properties for τ_2

Ranking functions and Invariants: Example

Transition system:

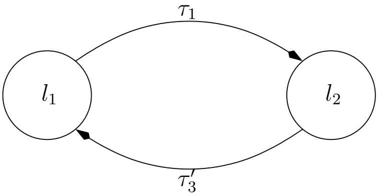


$$
\begin{array}{llllll}
\rho_{\tau_1}: & & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_2}: & y \geq 1, & y < z, & x' = x + 1, & y' = y, & z' = z - 1 \\
\rho_{\tau_3}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

and ranking function $f(x, y, z) = z$, fulfiling all properties for τ_2
we can remove τ_2

Ranking functions and Invariants: Example

Transition system:



$$
\begin{array}{lllllll}
\rho_{\tau_1}: & & y \geq 1, & x' = x - 1, & y' = y, & & z' = z \\
\rho_{\tau_3'}: & y \geq 1, & y \geq z, & x' = x, & & y' = x + y, & z' = z
\end{array}
$$

and ranking function $f(x, y, z) = z$, fulfiling all properties for τ_2
we can remove τ_2

Ranking functions and Invariants: Combined problem

In order to prove properties of the ranking function we may need to generate invariants.

Generation of both invariants and ranking functions should be combined in the same satisfaction problem.

Both are found at the same time [BMS2005].

Ranking functions and Invariants: Combined problem

In order to prove properties of the ranking function we may need to generate invariants.

Generation of both invariants and ranking functions should be combined in the same satisfaction problem.

Both are found at the same time [BMS2005].

In order to be correct we need to have two transition systems:

- the original system (extended with all found invariants) for invariant generation.
- the *termination transition system* which includes the transitions not yet proved to be terminating.

Similar to the *cooperation graph* in [BCF2013].

Our approach: Example

The approach in [BMS2005] is nice but in practice some problems arise:

- May need several invariants before finding a ranking function.

  We should be able to generate invariants even if there is no ranking function (how to guide the search?).

- Might be no ranking function fulfiling all properties

  We have to generate *quasi-ranking functions*.

  Similar concept as in e.g. Amir Ben-Amram's work.

  May not fulfil some of the properties.
  For instance, boundedness or decreasingness or even both.

Our approach: optimization vs satisfaction

Our solution:

Consider that this is an optimization problem
rather than a satisfaction problem

We want to get a ranking function but if it is not possible
we want to get as much properties as possible.

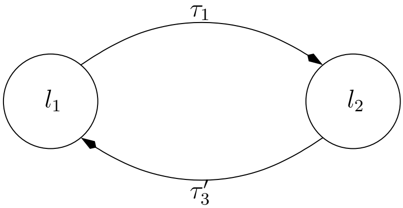
Use different weights to express which properties we prefer

Encode the problem using Max-SMT,

We use again Barcelogic to solve it.

Our approach: Example

Transition system:

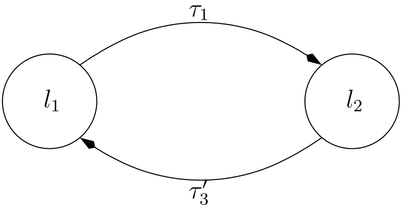


$$
\begin{array}{llllll}
\rho_{\tau_1}: & & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_3'}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

There is no ranking function that fulfils all conditions.

Our approach: Example

Transition system:



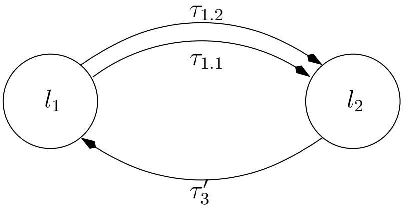
$$\begin{array}{lllll} \rho_{\tau_1}: & & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\ \rho_{\tau_3'}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z \end{array}$$

$f(x, y, z) = x$ is *non-increasing* and *strict decreasing* for τ_1.

However, it is not **bounded** (*soft*).

Our approach: Example

Transition system:



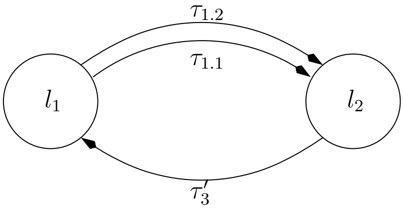
$$
\begin{array}{lllllll}
\rho_{\tau_{1.1}}: & x \geq 0 & y \geq 1, & x' = x - 1, & y' = y, & & z' = z \\
\rho_{\tau_{1.2}}: & x < 0 & y \geq 1, & x' = x - 1, & y' = y, & & z' = z \\
\rho_{\tau'_3}: & y \geq 1, & y \geq z, & x' = x, & & y' = x + y, & z' = z
\end{array}
$$

$f(x, y, z) = x$ is *non-increasing* and *strict decreasing* for τ_1.

However, it is not **bounded** (*soft*).

Our approach: Example

Transition system:



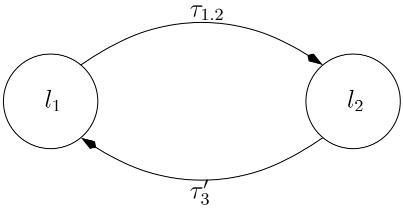
$$
\begin{array}{llllll}
\rho_{\tau_{1.1}}: & x \geq 0 & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_{1.2}}: & x < 0 & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau'_3}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

Now $f(x, y, z) = x$ is a ranking function for $\tau_{1.1}$

We can remove it!

Our approach: Example

Transition system:



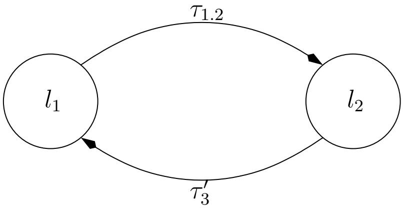
$$
\begin{array}{llllll}
\rho_{\tau_{1.2}}: & x < 0 & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\
\rho_{\tau_3'}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z
\end{array}
$$

Now $f(x, y, z) = x$ is a ranking function for $\tau_{1.1}$

We can remove it!

Our approach: Example

Transition system:

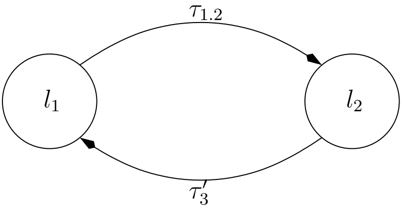


$$\begin{array}{lllllll} \rho_{\tau_{1.2}}: & x < 0 & y \geq 1, & x' = x - 1, & y' = y, & z' = z \\ \rho_{\tau_3'}: & y \geq 1, & y \geq z, & x' = x, & y' = x + y, & z' = z \end{array}$$

Finally, $f(x, y, z) = y$ is used to discard τ_3'.

Our approach: Example

Transition system:

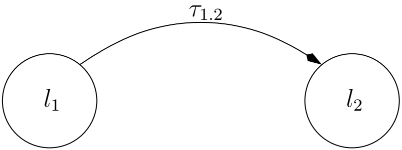


$$
\begin{array}{lllllll}
\rho_{\tau_{1.2}}: & x < 0 & y \geq 1, & x' = x - 1, & y' = y, & & z' = z \\
\rho_{\tau_3'}: & y \geq 1, & y \geq z, & x' = x, & & y' = x + y, & z' = z
\end{array}
$$

Finally, $f(x, y, z) = y$ is used to discard τ_3'.
But we need $x < 0$ in l_2, which is a *Termination Implication*

Our approach: Example

Transition system:



$$\rho_{\tau_{1.2}}: \quad x < 0 \quad y \geq 1, \quad x' = x - 1, \quad y' = y, \quad z' = z$$

Finally, $f(x, y, z) = y$ is used to discard τ_3'.
But we need $x < 0$ in l_2, which is a *Termination Implication*
We are DONE!

Contributions [Larraz,Oliveras,Rodríguez,Rubio 2013]

- A novel optimization-based method for proving termination.
- New inferred properties: Termnation Implications.
- No fixed number of supporting invariants *a priori*.
- Goal-oriented invariant generation.
- Progress in the absence of ranking functions (quasi-ranking functions).
- All these techniques have been implemented in CppInv

Experimental evaluation:

Two sources of benchmarks:

- coming from T2 (Microsoft Cambridge). Thanks!
- code made by undergraduate students taken from a programming learning environment Jutge.org

Experimental evaluation:

Two sources of benchmarks:

- coming from T2 (Microsoft Cambridge). Thanks!
- code made by undergraduate students taken from a programming learning environment Jutge.org In contrast to the standard academic examples the code is:
  - involved and ugly
  - unnecessary conditional statements
  - includes repeated code

Experimental evaluation:

	#ins.	CppInv	T2
Set1	449	238	245
Set2	472	276	279

Table: Results with benchmarks from T2

	#ins.	CppInv	T2
P11655	367	324	328
P12603	149	143	140
P12828	783	707	710
P16415	98	81	81
P24674	177	171	168
P33412	603	478	371

	#ins.	CppInv	T2
P40685	362	324	329
P45965	854	780	793
P70756	280	243	235
P81966	3642	2663	926
P82660	196	174	177
P84219	413	325	243

Table: Results with benchmarks from Jutge.org.

Outline

Further work

Other problems where using the optimization (Max-SMT) approach can help:

- Application to non-termination analysis:
  Maximize the exit paths to be removed.
- Application to verification of program postconditions (after loops)
  Maximize the properties that are ensured.
- Application to invariant generation in sequences of loops
  Make the initiation condition *soft* and if it is not fulfiled, use it as postcondition of the previous loop.

  Might be important for scalability!

Further work

- Apply our techniques to program synthesis
- Prove non-termination.
- Combine termination and non-termination proofs.
- Improve the non-linear arithmetic solver and the interaction with the invariant generation and termination engine.
- Consider other program properties

Conclusions

Two main conclusions:

- Using SMT and Max-SMT automatic invariant generation and termination proving become feasible.
- In constraint-based program analysis it is often better to consider that we have optimization problems rather than satisfaction problems!

Thank you!