Compositional Program Analysis using Max-SMT

Albert Rubio

Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, José Miguel Rivero and Enric Rodríguez-Carbonell

Universitat Politècnica de Catalunya - Barcelona Tech

UCM Seminar
March 2018

Overview of the talk

Overview of the talk

Motivation

- **Main Goal:** Build static analysis tools for programmers.
  - Fully automatic.
  - Efficient.
  - Scalable.

Motivation

- **Main Goal:** Build static analysis tools for programmers.
  - Fully automatic.
  - Efficient.
  - Scalable.

- **Strategy:** Take advantage of powerful arithmetic constraint solvers.

SMT solvers

Constraint-based Program Analysis techniques

Motivation

- **Main Goal:** Build static analysis tools for programmers.
  - Fully automatic.
  - Efficient.
  - Scalable.

- **Strategy:** Take advantage of powerful arithmetic constraint solvers.

Max-SMT solvers

Constraint-based Program Analysis techniques

Motivation

- **Main Goal:** Build static analysis tools for programmers.
  - Fully automatic.
  - Efficient.
  - Scalable.

- **Strategy:** Take advantage of powerful arithmetic constraint solvers.

Max-SMT solvers

Constraint-based Program Analysis techniques

Goal: Verify safety and liveness properties of programs

Motivation

- **Main Goal:** Build static analysis tools for programmers.
  - Fully automatic.
  - Efficient.
  - Scalable.

- **Strategy:** Take advantage of powerful arithmetic constraint solvers.

Max-SMT solvers

Constraint-based Program Analysis techniques

Goal: Verify safety and liveness properties of programs

Challenge: discover (loop) invariants.

How can we guide the search?

Overview of the talk

SMT solvers

We make extensive use of SMT solvers inside our program analysis tools.

SAT and SMT solvers gain efficiency by:

- addressing only (expressive enough) decidable fragments of a certain logic
- incorporate domain-specific reasoning, e.g:
  - arithmetic reasoning
  - equality
  - data structures (arrays, lists, stacks, ...)

- SAT: use propositional logic as the formalization language
  - \+ high degree of efficiency
  - \- expressive (all NP-complete) but involved encodings

- SMT: propositional logic + domain-specific reasoning
  - \+ improves the expressivity
  - \- certain (but acceptable) loss of efficiency

Need and Applications of SMT

- Some problems are more naturally expressed in other logics than propositional logic, e.g:
  - Software verification needs reasoning about equality, arithmetic, data structures, ...
- SMT consists of deciding the satisfiability of a (ground) FO formula with respect to a background theory
- Example (Equality with Uninterpreted Functions – EUF):

$$g(a)=c \quad \wedge \quad (\ f(g(a))\neq f(c) \ \vee \ g(a)=d\ ) \quad \wedge \quad c\neq d$$

- Wide range of applications:
  - Predicate abstraction
  - Model checking
  - Scheduling
  - Test generation
  - ...

Theories of Interest - Arithmetic

- Very useful for obvious reasons
- Restricted fragments support more efficient methods:
  - Bounds: $x \bowtie k$ with $\bowtie \in \{<, >, \leq, \geq, =\}$
  - Difference logic: $x - y \bowtie k$, with $\bowtie \in \{<, >, \leq, \geq, =\}$
  - UTVPI: $\pm x \pm y \bowtie k$, with $\bowtie \in \{<, >, \leq, \geq, =\}$
  - Linear arithmetic, e.g: $2x - 3y + 4z \leq 5$
  - Non-linear arithmetic, e.g: $2xy + 4xz^2 - 5y \leq 10$
  - Variables are either reals or integers

SMT problems

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

Question: Is there any *interpretation (solution)* 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 problems

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

Question: Is there any *interpretation (solution)* 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 problems

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

Question: Is there any *interpretation (solution)* that satisfies the formula?

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

$$(x < 0 \lor x \leq y \lor \underline{y < z}) \land (\underline{x \geq 0}) \land (x > y \lor \underline{y < z})$$

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

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

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.

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

Non-linear SMT solving

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

Question: Is there any solution that satisfies the formula?

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

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

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

Non-linear SMT solving

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

Question: Is there any solution 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 SMT solving

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

Question: Is there any solution 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 (Hilbert's 10th problem).
- *Reals:* decidable (Tarski) **but** algorithms have prohibitive complexity.

Non-linear SMT solving

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

Question: Is there any solution 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 (Hilbert's 10th problem).
- *Reals:* decidable (Tarski) **but** algorithms have prohibitive complexity.

Incomplete solvers focus on either satisfiability or unsatisfiability.

Non-linear SMT solving

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

Question: Is there any solution 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 (Hilbert's 10th problem).
- *Reals:* decidable (Tarski) **but** algorithms have prohibitive complexity.

Incomplete solvers focus on either satisfiability or unsatisfiability.

Solving non-linear SMT formulas

- Need to handle large formulas with non-linear arithmetic and *complex* boolean structure.
- Barcelogic has shown to be the best SMT-solver proving satisfiability of this kind of problems.
- Barcelogic can handle Max-SMT formulas (over non-linear arithmetic) as well.

Overview of the talk

Invariant generation

Definition

An <u>invariant</u> of a program at a location is an assertion over the program variables that remains true whenever the location is reached.

Invariant generation

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.

Invariant generation

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 focus on inductive invariants.

Constraint-based invariant generation

We inspire ourselves with the constraint-based method [CSS'03].

Assume input programs consist of linear expressions.

Constraint-based invariant generation

We inspire ourselves with the constraint-based method [CSS'03].

Assume input programs consist of linear expressions.

Keys:

- Use a template for candidate invariants.

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

Constraint-based invariant generation

We inspire ourselves with the constraint-based method [CSS'03].

Assume input programs consist of linear expressions.

Keys:

- Use a template for candidate invariants.

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

- Impose initiation and consecution conditions obtaining an $\exists\forall$ problem over non-linear arithmetic.

Constraint-based invariant generation

We inspire ourselves with the constraint-based method [CSS'03].

Assume input programs consist of linear expressions.

Keys:

- Use a template for candidate invariants.

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

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

Scalar invariant generation: Example

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:  c1 a + c2 s + c3 t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

▸ Back

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:  c1 a + c2 s + c3 t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

$\exists\ c_1,\ c_2,\ c_3,\ d\ \ \forall\ a,\ s,\ t$

$\underbrace{true \Longrightarrow c_1 \cdot 0 + c_2 \cdot 1 + c_3 \cdot 1 + d \leq 0}\ \wedge$ Initiation condition

$$\underbrace{s \leq N \wedge c_1 \cdot a + c_2 \cdot s + c_3 \cdot t + d \leq 0 \Longrightarrow c_1 \cdot (a+1) + c_2 \cdot (s+t+2) + c_3 \cdot (t+2) + d \leq 0}_{\text{consecution condition}}$$

Scalar invariant generation: Example

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:   c1a + c2s + c3t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

$\exists\, c_1,\ c_2,\ c_3,\ d\ \ \forall\, a,\ s,\ t$

$\underbrace{c_2 + c_3 + d \leq 0}\ \wedge$ Initiation condition

$$\underbrace{s \leq N \wedge c_1 \cdot a + c_2 \cdot s + c_3 \cdot t + d \leq 0 \Longrightarrow c_1 \cdot a + c_2 \cdot s + (c_2 + c_3) \cdot t + c_1 + 2c_2 + 2c_3 + d \leq 0}_{\text{consecution condition}}$$

Scalar invariant generation: Example

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:   c1a + c2s + c3t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

Apply Farkas' Lemma to remove $\forall\, a, s, t$

Use Barcelogic to solve the non-linear SMT problem!

Back

Scalar invariant generation: Example

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:  c1 a + c2 s + c3 t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

Apply Farkas' Lemma to remove $\forall\ a,\ s,\ t$

Use Barcelogic to solve the non-linear SMT problem!

$$\{c_1 = -2, c_2 = 0, c_3 = 1, d = -1\}$$

Back

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:  −2a + 0s + 1t − 1 ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

Apply Farkas' Lemma to remove $\forall\ a,\ s,\ t$

Use Barcelogic to solve the non-linear SMT problem!

$$\{c_1 = -2, c_2 = 0, c_3 = 1, d = -1\}$$

Back

Scalar invariant generation: Example

Square root of a natural number N:

```
int isqrt(int N) { //integer square root
  int a = 0, s = 1, t = 1;
  // Inv:  t ≤ 2a + 1
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

Apply Farkas' Lemma to remove $\forall\ a,\ s,\ t$

Use Barcelogic to solve the non-linear SMT problem!

$$\{c_1 = -2, c_2 = 0, c_3 = 1, d = -1\}$$

Back

Program analysis using SMT and Max-SMT

We have used this approach for:

- Array invariant generation. [VMCAI2013]
- Termination analysis using Max-SMT. [FMCAD2013] (inspired by [BMS'05])

  Key notion: quasi-ranking functions
- Non-Termination analysis using Max-SMT. [CAV2014]

  Key notion: quasi-invariant/conditional invariants

Overview of the talk

Safety verification

Aim: verify assertions in large programs (several consecutive loops).

Our approach: Goal oriented. Starts from the postcondition.
Automatically generate intermediate assertions!!

Simple example:

```
while (j>0) {
  j--;
  i++;
}

while (i>0) {
    x=x+5;
    i--;
}
assert(x≥0);
```

Safety verification

Aim: verify assertions in large programs (several consecutive loops).

Our approach: Goal oriented. Starts from the postcondition.
Automatically generate intermediate assertions!!

Simple example:

```
while (j>0) {
  j--;
  i++;
}
assert(x + 5*i >=0);
while (i>0) {
    x=x+5;
    i--;
}
assert(x>=0);
```

Safety verification

Aim: verify assertions in large programs (several consecutive loops).

Our approach: Goal oriented. Starts from the postcondition.
Automatically generate intermediate assertions!!

Simple example:

```
assert(j>=0 and x + 5*(i+j) >=0);
while (j>0) {
  j--;
  i++;
}
assert(x + 5*i >=0);
while (i>0) {
    x=x+5;
    i--;
}
assert(x≥0);
```

Conditional invariant generation

Definition

A formula is a conditional (inductive) invariant at a program location if:

- Consecution condition holds.

Conditional invariant generation

Definition

A formula is a conditional (inductive) invariant at a program location if:

- Consecution condition holds.

- but Initiation condition may not hold.

Conditional invariant generation

Definition

A formula is a conditional (inductive) invariant at a program location if:

- Consecution condition holds. Hard
- but Initiation condition may not hold.

Conditional invariant generation

Definition

A formula is a conditional (inductive) invariant at a program location if:

- Consecution condition holds. Hard
- but Initiation condition may not hold. Soft

Key: We prefer invariants but we can live with conditional invariants

Conditional invariant generation

Altogether we have:

- Initiation condition (soft)
- Consecution condition (hard)

Conditional invariant generation

Altogether we have:

- Initiation condition (soft)
- Consecution condition (hard)
- Plus implication of the Postcondition (hard)

Conditional invariant generation

Altogether we have:

- Initiation condition (soft)
- Consecution condition (hard)
- Plus implication of the Postcondition (hard)

Solve the problem with a Max-SMT solver (we use Barcelogic)

Conditional invariant generation

Altogether we have:

- Initiation condition (soft)
- Consecution condition (hard)
- Plus implication of the Postcondition (hard)

Solve the problem with a Max-SMT solver (we use Barcelogic)

If initiation condition holds we are done

Conditional invariant generation

Altogether we have:

- Initiation condition (soft)
- Consecution condition (hard)
- Plus implication of the Postcondition (hard)

Solve the problem with a Max-SMT solver (we use Barcelogic)

If initiation does not hold we have a new Postcondition for previous code

Conditional invariant generation

Altogether we have:

- Initiation condition (soft)
- Consecution condition (hard)
- Plus implication of the Postcondition (hard)

Solve the problem with a Max-SMT solver (we use Barcelogic)

If initiation does not hold we have a new Postcondition for previous code

call recursively to the safety checker

Safety verification: Recovering from failures

In case of failure of the recursive call to the safety checker

- Add the negation of the conditional invariant in the corresponding locations: *Narrow the loop*
- Try to prove the Postcondition again (with more information).

Narrowing loops: Recovering from failures

Simple example:

```
int x=-50;
int y=nondet();


while (x<0) {
  x = x + y;
  y = y + 1;
}

assert(y>0);
```

Skip

Safety verification

Narrowing loops: Recovering from failures

Simple example:

```
int x=-50;
int y=nondet();

assert(y>0);
while (x<0) {
  x = x + y;
  y = y + 1;
}

assert(y>0);
```

Skip

Safety verification

Narrowing loops: Recovering from failures

Simple example:

```
int x=-50;
int y=nondet();

assert(y>0); // Conditional invariant
while (x<0) {
  x = x + y;
  y = y + 1;
}

assert(y>0);
```

Skip

Safety verification

Narrowing loops: Recovering from failures

Simple example:

```
int x=-50;
int y=nondet();
assume(!(y>0));

while (x<0) {
  assume(!(y>0));
  x = x + y;
  y = y + 1;
}
assert(y>0);
```

▸ Skip

Narrowing loops: Recovering from failures

Simple example:

```
int x=-50;
int y=nondet();
assume(y<=0);

while (x<0) {
  assume(y<=0);
  x = x + y;
  y = y + 1;
}
assert(y>0);
```

Skip

Safety verification

Narrowing loops: Recovering from failures

Simple example:

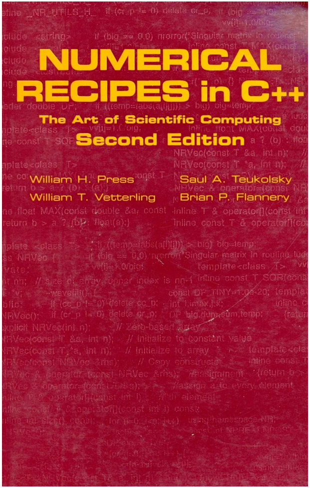
```
int x=-50;
int y=nondet();
assume(y<=0);
assert(x<0); // Invariant
while (x<0) {
  assume(y<=0);
  x = x + y;
  y = y + 1;
}
assert(y>0); // Unreachable!
```

Skip

Experiments

Our techniques have been implemented in a tool called VeryMax

- 217 programs taken from *Numerical Recipes in C++*
- up to 284 lines of code
- 6452 safety problems
- 6106 can be proved
- highly parallelizable
- FMCAD 2015



Conditional Termination

Given a program (loop), obtain a (pre-)condition ensuring its termination.

Conditional Termination

Given a program (loop), obtain a (pre-)condition ensuring its termination.

Our approach:

- Find ranking functions using (linear) templates
- Find supporting conditional invariants (like before)

Conditional Termination

Given a program (loop), obtain a (pre-)condition ensuring its termination.

Our approach:

- Find ranking functions using (linear) templates
- Find supporting conditional invariants (like before)

Encode the problem with Max-SMT

Conditional Termination

Given a program (loop), obtain a (pre-)condition ensuring its termination.

Our approach:

- Find ranking functions using (linear) templates
- Find supporting conditional invariants (like before)

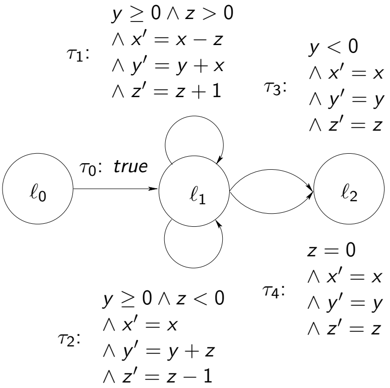
Encode the problem with Max-SMT

Using conditional termination we can

- prove termination by cases. Skip
- combine termination and non-termination analysis (in parallel).
- prove termination of consecutive loops in a compositional way.

Running example

```
int main() {
  int x, y, z;
  x = nondet();
  y = nondet();
  z = nondet();
  while (y ≥ 0 && z ≠ 0) {
    if (z < 0) { y = y + z;
                 z = z − 1;
    } else { x = x − z;
             y = y + x;
             z = z + 1;
    }
  }
}
```



Ranking functions and Conditional 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)

Use a linear template for the ranking function as well.

Ranking functions and Invariants: Combined problem

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

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

Back

Ranking functions and Invariants: Combined problem

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

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

Back

1. $\mathcal{I} \wedge \tau_i \Longrightarrow f(x_1, \ldots, x_n) \geq 0$ (bounded)
2. $\mathcal{I} \wedge \tau_i \Longrightarrow f(x_1, \ldots, x_n) > f(x'_1, \ldots, x'_n)$ (strict-decreasing)
3. $\mathcal{I} \wedge \tau_j \Longrightarrow f(x_1, \ldots, x_n) \geq f(x'_1, \ldots, x'_n)$ 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 conditional invariants and ranking functions should be combined in the same optimization problem. ▸ Back

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

Considering conditional invariants give more chances to the solver

Ranking functions and Invariants: Combined problem

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

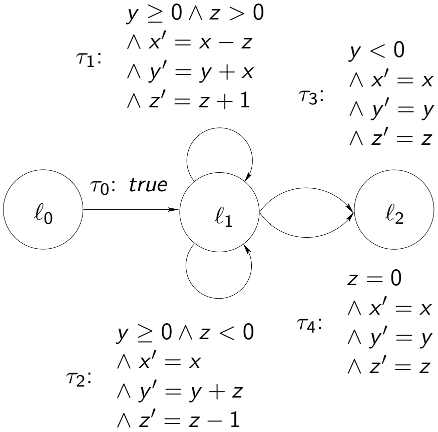
Generation of both conditional invariants and ranking functions should be combined in the same optimization problem. ▸ Back

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

Considering conditional invariants give more chances to the solver

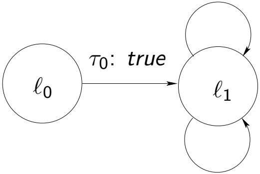
But we get a conditional termination proof

Running example



Running example

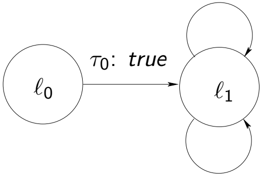
$$
\tau_1: \quad \begin{array}{l} y \geq 0 \wedge z > 0 \\ \wedge\, x' = x - z \\ \wedge\, y' = y + x \\ \wedge\, z' = z + 1 \end{array}
$$



$$
\tau_2: \quad \begin{array}{l} y \geq 0 \wedge z < 0 \\ \wedge\, x' = x \\ \wedge\, y' = y + z \\ \wedge\, z' = z - 1 \end{array}
$$

Running example

$$\tau_1: \quad \begin{array}{l} y \geq 0 \wedge z > 0 \\ \wedge\ x' = x - z \\ \wedge\ y' = y + x \\ \wedge\ z' = z + 1 \end{array}$$

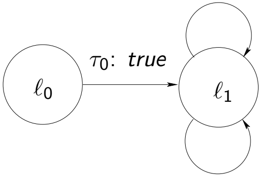


$$\tau_2: \quad \begin{array}{l} y \geq 0 \wedge z < 0 \\ \wedge\ x' = x \\ \wedge\ y' = y + z \\ \wedge\ z' = z - 1 \end{array}$$

- $z < 0$ is a conditional invariant at location ℓ_1
- y is a ranking function
  1. τ_1 is disabled
  2. τ_2 is bounded and strictly decreasing

Running example

$$\tau_1: \quad \begin{array}{l} y \geq 0 \wedge z > 0 \\ \wedge\ x' = x - z \\ \wedge\ y' = y + x \\ \wedge\ z' = z + 1 \end{array}$$

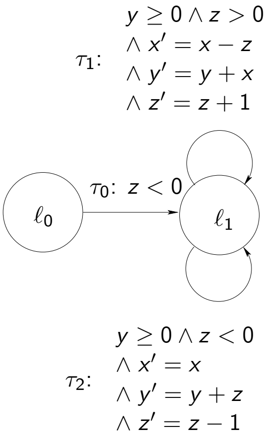


$$\tau_2: \quad \begin{array}{l} y \geq 0 \wedge z < 0 \\ \wedge\ x' = x \\ \wedge\ y' = y + z \\ \wedge\ z' = z - 1 \end{array}$$

We have a conditional proof:

The system terminates if the condition $z < 0$ holds at ℓ_0

Running example



We have a conditional proof:

The system terminates if the condition $z < 0$ holds at ℓ_0 (or τ_0)

Running example: Narrowing

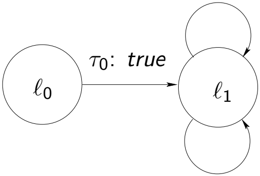
In order to complete the termination proof we have to consider the complementary problem.

Narrow the transitions removing all states that we already know that are terminating.

We can do better than just add the negation of the condition in the entry.

Running example: Narrowing

$$\tau_1: \quad \begin{array}{l} y \geq 0 \wedge z > 0 \\ \wedge\ x' = x - z \\ \wedge\ y' = y + x \\ \wedge\ z' = z + 1 \end{array}$$



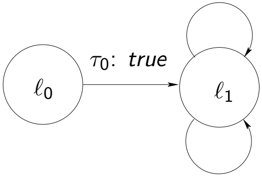
$$\tau_2: \quad \begin{array}{l} y \geq 0 \wedge z < 0 \\ \wedge\ x' = x \\ \wedge\ y' = y + z \\ \wedge\ z' = z - 1 \end{array}$$

We know more!:

whenever $z < 0$ holds at ℓ_1 the system terminates

Running example: Narrowing

$$\tau_1\colon \quad \begin{array}{l} y \geq 0 \wedge z > 0 \\ \wedge\ x' = x - z \\ \wedge\ y' = y + x \\ \wedge\ z' = z + 1 \end{array}$$



$$\tau_2\colon \quad \begin{array}{l} y \geq 0 \wedge z < 0 \\ \wedge\ x' = x \\ \wedge\ y' = y + z \\ \wedge\ z' = z - 1 \end{array}$$

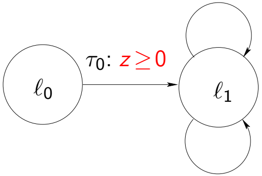
Narrow the transition system according to this:

whenever $z < 0$ holds at ℓ_1 the system terminates

Running example: Narrowing

$z \geq 0$

$$\tau_1: \quad \begin{array}{l} y \geq 0 \land z > 0 \\ \land\, x' = x - z \\ \land\, y' = y + x \\ \land\, z' = z + 1 \end{array}$$



$z \geq 0$

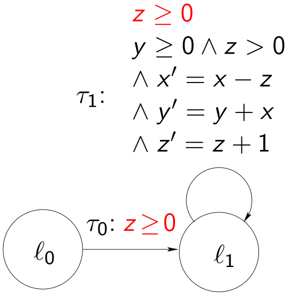
$$\tau_2: \quad \begin{array}{l} y \geq 0 \land z < 0 \\ \land\, x' = x \\ \land\, y' = y + z \\ \land\, z' = z - 1 \end{array}$$

Narrow the transition system according to this:

whenever $z < 0$ holds at ℓ_1 the system terminates

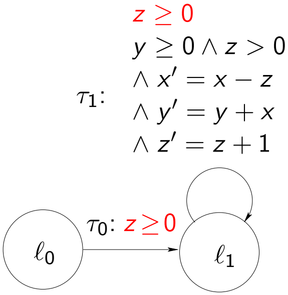
Running example. Narrowing

After simplifying the transition system we get:



Running example. Narrowing

After simplifying the transition system we get:

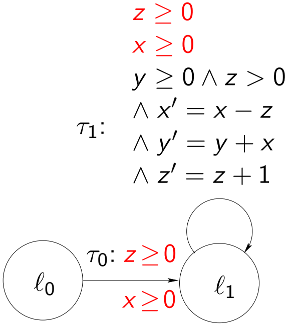


Conditionally terminates:

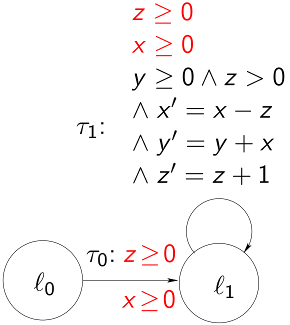
- $x < 0$ is a conditional invariant at location ℓ_1
- y is a ranking function
  1. τ_1 is bounded and strictly decreasing

Running example. Narrowing

Narrowing again with the complement of $x < 0$ we get:



Narrowing again with the complement of $x < 0$ we get:



Which terminates with x as a ranking function

Compositional Termination Analysis

Aim: prove termination of large programs (several consecutive loops).

New approach:

1. Obtain a conditional termination proof.
2. Check the condition as a Safety property.

Simple example:

```
assume(x > y && y ≥ 0);
while (y > 0) {
  x = x - 1;
  y = y - 1;
}
while (y < 0) {
  y = y + x;
}
```

Compositional Termination Analysis

Aim: prove termination of large programs (several consecutive loops).

New approach:

1. Obtain a conditional termination proof.
2. Check the condition as a Safety property.

Simple example:

```
assume(x > y && y ≥ 0);
while (y > 0) {
  x = x - 1;
  y = y - 1;
}
assert(x > 0); Rank: -y
while (y < 0) {
  y = y + x;
}
```

Compositional analysis through Conditional Termination

Aim: verify termination in large programs (several consecutive loops).

New approach:

1. Obtain a conditional termination proof.
2. Check the condition as a Safety property.

Compositional analysis through Conditional Termination

Aim: verify termination in large programs (several consecutive loops).

New approach:

1. Obtain a conditional termination proof.
2. Check the condition as a Safety property.
3. In case of failure of the Safety checker

   Narrow the loop and try again!

Compositional analysis through Conditional Termination

Aim: verify termination in large programs (several consecutive loops).

New approach:

1. Obtain a conditional termination proof.
2. Check the condition as a Safety property.
3. In case of failure of the Safety checker

   Narrow the loop and try again!

We can handle every loop independently

Experiments

Our techniques have been implemented in VeryMax

Results:

- won the C Integer programs categories of the Terminaton Competition in 2016 and 2017

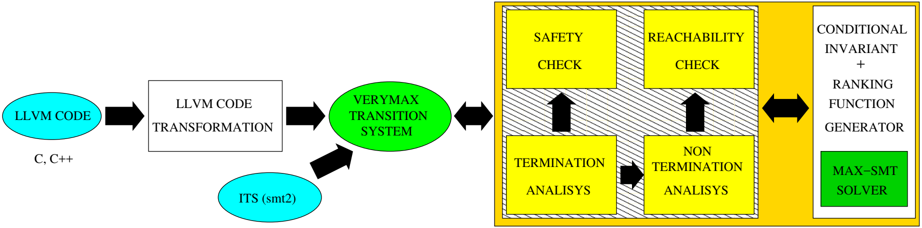
- Comparison with tools at the Termination category of SVComp. On the 358 benchmarks not involving recursion or pointers (273)

Tool	Term	NTerm	Fail	TO	Total (s)
AProVE	222	76	41	19	10235.44
SeaHorn	189	75	22	72	34760.69
UltimateBuchiA	224	103	25	6	7882.13
VeryMax	231	101	26	0	2444.29

See our paper at TACAS 2017 for more details.

Overview of the talk

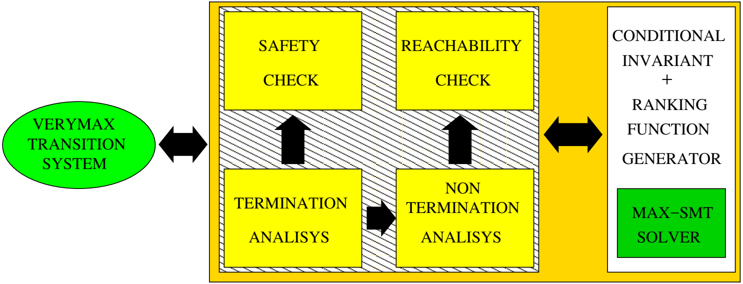
VeryMax global architecture



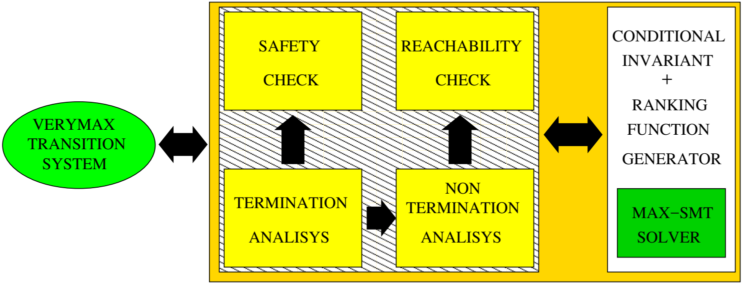
Two phases

1. Front-end. From source programs to VeryMax Transition Systems
2. Static Analysis Tools

VeryMax static analysis tools



VeryMax static analysis tools



VeryMax can

1. check safety properties
2. check reachability properties
3. prove termination
4. prove non-termination

Overview of the talk

Conclusions

Two main conclusions:

- Using SMT and Max-SMT, automatic generation of needed (conditional) invariants can be made efficiently.
- Scalable program verification becomes feasible

Other potential applications of conditional analysis and Max-SMT:

- Analysis of concurrent/distributed systems.
- Program synthesis.
- Program repair (minimize changes).
- Using Max-SMT we can express preferences among possible solutions

Thank you!