Scalable Program analysis using Max-SMT

Albert Rubio

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

Universitat de Vic
Universitat Politècnica de Catalunya - Barcelona Tech
Microsoft Research, Cambridge

supported by MINECO/FEDER project TIN2015-69175-C4-3-R

University of Bergen
October 2016

Overview of the talk

Overview of the talk

The Halting Problem

The Halting Problem

The longer it keeps you waiting
the more you appreciate a termination analysis

Software Reliability

- Safety Critical Software.
  - Avionics (= aviation + electronics)
  - Railway systems
  - Automotive
  - Drone software
  - Health care

  There are international software safety standards that need to be met.
- Software in business.
- Web services
- ...

Program Verification

Reasoning about software correctness goes back to the early ages of computer science:

Turing (1949), Floyd (1967), Hoare (1969), Dijkstra (1976)

Prove formally that

- The program terminates
  All executions traces are finite (halting problem)
- The program meets a given specification
  - For all possible inputs (not just testing some inputs)
  - For a property given in some specification language

Both problems are undecidable even for quite simple programming languages and specification languages.

Specification language

Hoare logic: Pre/Post specifications

Specification language

Hoare logic: Pre/Post specifications

Preconditions and Postconditions are written in First-order logic.
For instance:

- $0 \leq i \leq n - 1$
- $\forall\alpha : 1 \leq \alpha \leq n - 1 : v[\alpha - 1] \leq v[\alpha]$

A property (condition) is required to hold in some point of the program

It is the standard specification language for sequential programs.

Safety and liveness properties

- A safety property states that nothing bad happens

  For instance, in a system no ERROR/STOP state is reachable.

- A liveness property states that something good eventually happens

  For instance, in a system an action is eventually executed (fairness).

Safety and liveness properties are dual.

Safety and liveness properties

- A safety property states that nothing bad happens

  For instance, in a system no ERROR/STOP state is reachable.

Approaches to formal verification

- Deductive verification
- Model Checking
- Testing

Approaches to formal verification

Deductive verification

Approaches to formal verification

Deductive verification

- Given a system and its specification (and maybe other annotations).
- Mathematical proof obligations (theorems) are generated.
- These theorems are proved using:
  - Proof assistants (Isabelle, Coq, etc)
  - Theorem provers (Vampire, Spass, etc)
  - Satisfiability modulo theories (SMT) solvers (Z3, CVC4, Barcelogic, etc)

Approaches to formal verification

Deductive verification

- Given a system and its specification (and maybe other annotations).
- Mathematical proof obligations (theorems) are generated.
- These theorems are proved using:
  - Proof assistants (Isabelle, Coq, etc)
  - Theorem provers (Vampire, Spass, etc)
  - Satisfiability modulo theories (SMT) solvers (Z3, CVC4, Barcelogic, etc)

Trade-off between automation and both scalability and efficiency.

It also depends on the expressivity of the specification language.

A particular example of this approach is SPARK 2014

SPARK 2014

- SPARK is a programming language based on Ada.
- Ada is a general-purpose languages that was designed from the start (1983) with reliability, safety, and security in mind.
- SPARK is a specialized subset of Ada designed to facilitate the use of formal methods.
- SPARK is intended for applications that demand safety or security integrity.
- SPARK 2014 is a subset of Ada 2012
- SPARK 2014 is developed by Altran and AdaCore Companies (started at the University of Southampton).

SPARK 2014

Inherits from Ada:

- Powerful type system
- Automatically inserts runtime checks. For instance,
  - Array bounds check, Integer overflows, Divisions by zero
- Since Ada 2012, contract-based programming.
  Most common: Pre and Post conditions and loop invariants

```
procedure Increase (X : in out Integer) with
  Pre => X <= Max,
  --  It is the responsibility of every caller of Increase to check that
  --  its argument is less than Max.
  Post => X > X'Old;
  --  It is the responsibility of Increase's implementation to ensure that
  --  the returned value of X is  strictly greater than its initial value.
```

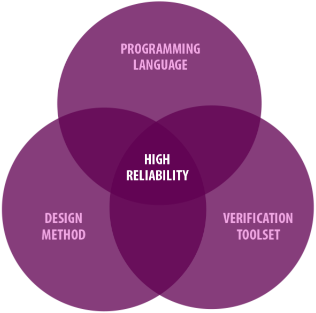
Does not include from Ada:

- pointers (but addresses are allowed), goto statement, exception handling, ...

SPARK 2014

Adds

- supports formal verification as well
  - proving safety (or security) properties
  - proving the software implementation meets a formal specification



Overview of the talk

SPARK 2014 Automation

SPARK 2014 intends to provide automatic verification of safety properties

But it may fail!

Need of loop invariants

Cannot be generated automatically
Weakness: it is not an easy task for developers!

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.

Deductive verification tools normally focus on inductive invariants.

Motivation

Our Main Goal: Build verification tools for programmers that are

- Fully automatic.
- Efficient.
- Scalable.

Motivation

Our Main Goal: Build verification tools for programmers that are

- Fully automatic.
- Efficient.
- Scalable.

Strategy: Take advantage of powerful arithmetic constraint solvers.

SMT solvers

Constraint-based Program Analysis techniques

Motivation

Our Main Goal: Build verification tools for programmers that are

- Fully automatic.
- Efficient.
- Scalable.

Strategy: Take advantage of powerful arithmetic constraint solvers.

Max-SMT solvers

Constraint-based Program Analysis techniques

Motivation

Our Main Goal: Build verification tools for programmers that are

- Fully automatic.
- Efficient.
- Scalable.

Strategy: Take advantage of powerful arithmetic constraint solvers.

Max-SMT solvers

Constraint-based Program Analysis techniques

Today's Goal: Verify safety properties of programs

Motivation

Our Main Goal: Build verification tools for programmers that are

- Fully automatic.
- Efficient.
- Scalable.

Strategy: Take advantage of powerful arithmetic constraint solvers.

Max-SMT solvers

Constraint-based Program Analysis techniques

Today's Goal: Verify safety properties of programs

Challenge: discover (loop) invariants.

How to guide the search?
How to make it scalabe?

Overview of the talk

SMT solvers

SAT and SMT (Satisfiability modulo theories) 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, like software verification, need reasoning about equality, arithmetic, data structures, ...
- 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:
  - Deductive verification
  - Model checking
  - Test case generation
  - Scheduling
  - ...

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 $\bowtie \in \{<, >, \leq, \geq, =\}$
  - 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 solving

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

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

Question: Is there asolution 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

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

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

Question: Is there a solution that satisfies the formula?

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

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

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

SMT solving

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

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

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

SMT solving

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

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

Question: Is there asolution 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.

SMT solving

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

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

Question: Is there asolution 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.

SMT solving

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

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

Question: Is there asolution that satisfies the formula?

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

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

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

- 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.

Optimization problems

(Weighted) Max-SMT problem

Input: Given an SMT formula $\varphi = C_1 \wedge \ldots \wedge C_m$, where some of the conditions 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.

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

Optimization problems

(Weighted) Max-SMT problem

Input: Given an SMT formula $\varphi = C_1 \wedge \ldots \wedge C_m$, where some of the conditions 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.

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

Barcelogic can handle Max-SMT formulas as well.

Overview of the talk

Invariant generation

Definition (Recall)

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.

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.

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

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:  c1a + c2s + c3t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

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;
}
```

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

$\underbrace{true \Longrightarrow c_1 \cdot 0 + c_2 \cdot 1 + c_3 \cdot 1 + d \leq 0}\ \wedge$ 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:  c1 a + c2 s + c3 t + d ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

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

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

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;
}
```

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

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:  −2a + 0s + 1t − 1 ≤ 0
  while (s ≤ N) {
    a = a + 1;
    s = s + t + 2;
    t = t + 2;
  }
  return a;
}
```

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

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;
}
```

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

Overview of the talk

Safety verification

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

New 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).

New 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).

New 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 codition (soft)
- Consecution condition (hard)

Conditional invariant generation

Altogether we have:

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

Conditional invariant generation

Altogether we have:

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

Solve the problem with a Max-SMT solver

Conditional invariant generation

Altogether we have:

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

Solve the problem with a Max-SMT solver

If initiation condition holds we are done

Conditional invariant generation

Altogether we have:

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

Solve the problem with a Max-SMT solver

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

Conditional invariant generation

Altogether we have:

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

Solve the problem with a Max-SMT solver

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

call recursively to the safety checker

Safety verification: Recovering from failures

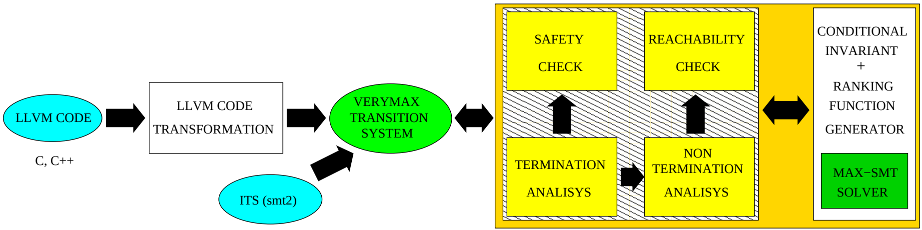
In case of failure of the recursive call to the safety checker

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

Overview of the talk

VeryMax global architecture

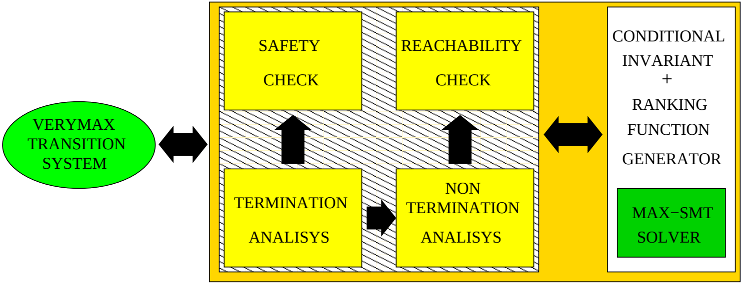
Our techniques have been implemented in a tool called VeryMax



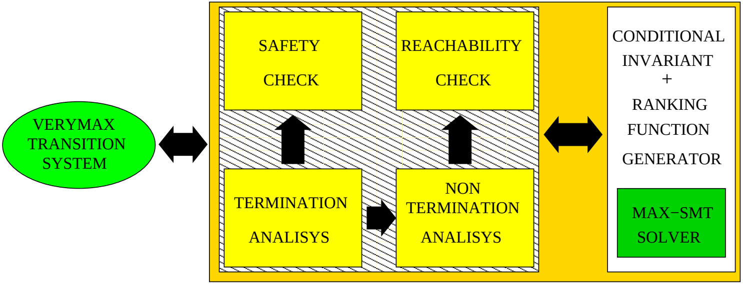
Two phases

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

Future developments:

- Reasoning with data structures
- Resource analysis

Thank you!