Program Verification Using

Automatic Generation of Invariants

Enric Rodríguez-Carbonell
Universitat Politècnica
de Catalunya
Barcelona

Deepak Kapur
University
of New Mexico
Albuquerque

Plan of the Talk

1. **Introduction**

2. **Invariant Generation**
   - General method for discovering invariants in loops without nesting
   - Conditions to compute strongest possible invariant expressible in assertion language
   - Instantiation for finding invariant polynomial equalities

3. **Program Verification**
   - Application with prototype verifier

4. **Future Work and Conclusions**

Introduction
Need for Software Verification

- **Critical systems**
  - safety
  - security
  - economy

- **Finding errors as soon as possible.** See Microsoft's *Software Productivity Tools* group: verification pays off

- **Main classes of techniques:**
  - Model checking
  - Abstract Interpretation
  - Theorem Proving

Introduction
Verification by Theorem Proving (1)

- Program behaviour specified by *pre-postconditions*
- Formulas called *verification conditions* encoding correctness of program are generated
- Verification conditions proved by *theorem prover*
- Program must also be annotated with *loop invariants*

loop invariants fundamental for program verification

Introduction
Verification by Theorem Proving (2)

$\{\texttt{Pre:}\ X \geq 0 \wedge Y \geq 0\,\}$

$x := X; y := Y; z := 0;$

$\{\texttt{Inv:}\ I(x, y, z, X, Y)\,\}$

```
while (y ≠ 0) do
     if (y mod 2 = 0) then
          x := 2 * x; y := y div 2;
     else
          y := y − 1; z := x + z;
     end if
end while
```

$\{\texttt{Post:}\ z = X * Y\,\}$

VERIFICATION CONDITIONS

- Precondition implies invariant
- Invariant preserved in `if` branch
- Invariant preserved in `else` branch
- Invariant implies postcondition

Introduction
Verification by Theorem Proving (3)

```
{Pre: X ≥ 0 ∧ Y ≥ 0 }
x := X; y := Y; z := 0;
{Inv: I(x, y, z, X, Y) }
while (y ≠ 0) do
    if (y mod 2 = 0) then
        x := 2 * x; y := y div 2;
    else
        y := y − 1; z := x + z;
    end if
end while
{Post: z = X * Y }
```

VERIFICATION CONDITIONS

- $X \geq 0 \wedge Y \geq 0 \Rightarrow I(X, Y, 0, X, Y))$
- $I(x, y, z, X, Y) \wedge y \,\textbf{mod}\, 2 = 0 \wedge y \neq 0 \Rightarrow I(2 * x, y \,\textbf{div}\, 2, z, X, Y)$
- $I(x, y, z, X, Y) \wedge y \,\textbf{mod}\, 2 = 1 \wedge y \neq 0 \Rightarrow I(x, y - 1, x + z, X, Y)$
- $I(x, y, z, X, Y) \wedge y = 0 \Rightarrow z = X * Y$

Introduction

Verification by Theorem Proving (4)

- **Problems** in the past
  1. Provers too weak and needed too much interaction
  2. Annotation burden: need to supply invariants by hand
- **Solutions** in the present
  1. Spectacular improvement of automated reasoning
  2. Automatic invariant generation

Plan of the Talk

1. **Introduction**
2. **Invariant Generation**
3. **Program Verification**
4. **Future Work and Conclusions**

Invariant Generation
Related Work (1)

- **Dynamic** approaches (e.g. Daikon) [Ernst et al'01.]
  1. Run programs on sample inputs
  2. Find candidate invariants from set of templates
- **Static** approaches
  1. Operate on program text
  2. Invariants found are correct
     - Top-down: *backward* propagation of *postcondition* [Müller-Olm & Seidl'03,'04]
     - Bottom-up: *forward* propagation of *precondition* (see next slide)
     - Constraint-based methods [Colón et al.'03, Sankaranarayanan et al.'04, Kapur'04]

Invariant Generation
Related Work (2)

Bottom-up approach

- Abstract Interpretation [Cousot & Cousot'77]
  - Interval ranges [Cousot & Cousot'76]
  - Linear inequalities [Cousot & Halbwachs'78]
  - Congruences [Granger'89,'91,Masdupuy'92]
  - Polynomial equalities
    [Rodriguez-Carbonell & Kapur'04,Colón'04]

- Difference Equations Method [Elspas et al.'72]
  1. Find program states using *difference equations*
  2. Eliminate loop counters

Invariant Generation

Overview

- **GOAL:** Generation of invariants for loops without nesting

```
while E(x̄) do
    if  C_1(x̄)  then x̄ := f_1(x̄);
    ...
    else if  C_i(x̄)  then x̄ := f_i(x̄);
    ...
    end if
end while
```

- **MEANS:** Forward propagation with acceleration
  - *Improves the number of iterations* before termination
  - *Reduces loss of information* involved in approximations
- **INSTANTIATION:** polynomial equalities as invariants

Invariant Generation

Forward Propagation (1)

$\{\, R(\bar{x}) \,\}$
```
while E(x̄) do
     if C_1(x̄) → x̄ := f_1(x̄);
     ...
     [] C_i(x̄) → x̄ := f_i(x̄);
     ...
     end if
end while
```

- Post-states of i-th branch, 1 step $Post_i(R)$:

$$Post_i(R) = \{\exists \bar{y}(\bar{x} = f_i(\bar{y}) \wedge R(\bar{y}) \wedge E(\bar{y}) \wedge C_i(\bar{y}))\}$$

- Post-states of i-th branch, any number of steps $Post_i^*(R)$:

$$Post_i^*(R) = \{ \bigvee_{s=1}^{\infty} \exists \bar{y}(\bar{x} = f_i^s(\bar{y}) \wedge R(\bar{y}) \wedge \bigwedge_{t=0}^{s-1} E(f_i^t(\bar{y})) \wedge C_i(f_i^t(\bar{y})))\}$$

Invariant Generation
Forward Propagation (2)

Fixpoint procedure for generating invariants

$R' := false$;

$R := R_0$;

`while` $R' \not\Leftrightarrow R$ `do`

  $R' := R$;

  $R := R \vee (\bigvee_{i=1}^{n} Post_i^*(R))$;

`end while`

Invariant Generation
Forward Propagation (3)

To make procedure realizable, in assertion language we need to:

- Over-approximate $\vee$
- Over-approximate $Post_i^*$
- Check equivalence of formulas

The procedure computes the *strongest invariant* if additionally

- There is strongest invariant in the language
- The over-approximation of $\vee$ is the best possible
- The over-approximation of $Post_i^*$ is the best possible

Invariant Generation
Forward Propagation (4)

To make procedure realizable, in assertion language we need to:

- Over-approximate $\vee$ $\longrightarrow$ **Disjunctively closed**
- Over-approximate $Post_i^*$ $\longrightarrow$ **Quantifier elimination**
- Check equivalence of formulas $\longrightarrow$ **Equivalence checking**

The procedure computes the *strongest invariant* if additionally

- There is strongest invariant in the language $\longrightarrow$ **Expressive**
- The over-approximation of $\vee$ is the best possible
- The over-approximation of $Post_i^*$ is the best possible

Invariant Generation

Polynomial Equalities as Invariants (1)

Instance of method: **conjunctions of polynomial equalities**

Satisfy all requirements:

- Over-approximation of $\vee$: intersection of ideals of polynomials exactly expresses $\vee$
- Over-approximation of $Post_i^*$: elimination of variables gives the best approximation possible
- Equivalence checking: computation of Gröbner bases
- Existence of strongest invariant: by Hilbert's basis theorem

Invariant Generation

Polynomial Equalities as Invariants (2)

$x := X; y := Y; z := 0;$

```
while (y ≠ 0) do
     if (y mod 2 = 0) then
          x := 2 * x; y := y div 2;
     else
          y := y - 1; z := x + z;
     end if
end while
```

TRACE

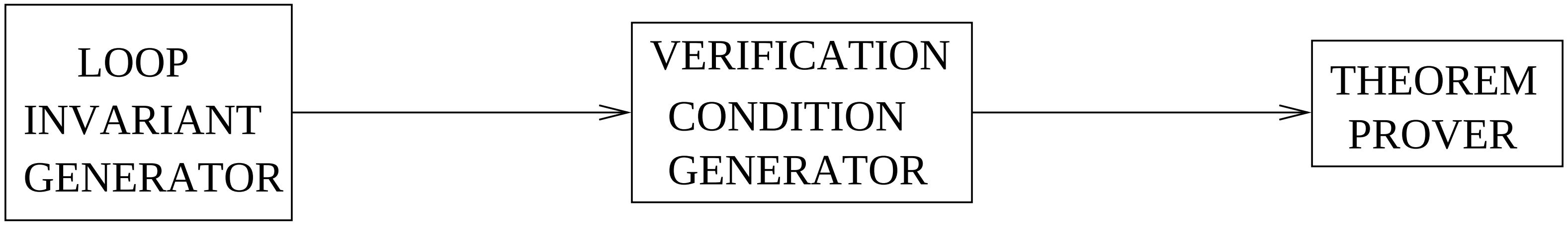
1. $x = X \wedge y = Y \wedge z = 0$
2. $x * z = X * z \ \wedge$
   $z + x * y = Y * X \ \wedge$
   $z * y * X + z^2 = z * Y * X$
3. $z + x * y = X * Y$
4. $z + x * y = X * Y$

Plan of the Talk

1. Introduction
2. Invariant Generation
3. **Program Verification**
4. Future Work and Conclusions

Program Verification

Scheme of an Implementation for the
Verification of Imperative Programs



Program Verification

Verification Condition Generator (1)

- **INPUT:** imperative programs annotated with pre/postconditions
- **PROGRAMMING LANGUAGE**
  - Integer variables
  - Arithmetic operators: $+$, $*$, **div**, **mod**
  - Procedure calls
- **SPECIFICATION LANGUAGE**
  - Untyped first-order logic with interpreted symbols
  - Interpreted function symbols: $+$, $*$, **div**, **mod**, **lcm**, **gcd**, **exp**
  - Interpreted predicate symbols: $=$, $\geq$, $>$

Program Verification

Verification Condition Generator (2)

$\{\texttt{Inv:}\, z + x * y = X * Y\,\}$
$\{\, y \neq 0 \wedge y\,\textbf{mod}\,2 = 0 \Rightarrow z + (2 * x) * (y\,\textbf{div}\,2) = X * Y\,\}$
$\texttt{while}\ (y \neq 0)\ \texttt{do}$
$\quad \{\, y\,\textbf{mod}\,2 = 0 \Rightarrow z + (2 * x) * (y\,\textbf{div}\,2) = X * Y\,\}$
$\quad \texttt{if}\ (y\,\textbf{mod}\,2 = 0)\ \texttt{then}$
$\qquad \{\, z + (2 * x) * (y\,\textbf{div}\,2) = X * Y\,\}$
$\qquad x := 2 * x;$
$\qquad \{\, z + x * (y\,\textbf{div}\,2) = X * Y\,\}$
$\qquad y := y\,\textbf{div}\,2;$
$\qquad \{\, z + x * y = X * Y\,\}$
$\quad \texttt{else} \qquad \cdots$
$\texttt{end while}$
$\{\, z + x * y = X * Y\,\}$

VERIFICATION CONDITION

$$z + x * y = X * Y \ \wedge\ y \neq 0 \ \wedge\ y\,\textbf{mod}\,2 = 0 \Rightarrow$$
$$z + (2 * x) * (y\,\textbf{div}\,2) = X * Y$$

Program Verification
Theorem Prover (1)

- First-order general theorem prover SPASS unsatisfactory
- Better results with our *ad-hoc* prover
  - Simplification: rewriting until getting $true$
  - Conditional rewriting $\longrightarrow$ strategies for proving theorems
  - Knowledge incorporated about integer numbers
  - Implemented in Prolog

Program Verification
Theorem Prover (2)

$$z + x * y = X * Y \land y \neq 0 \land y \,\mathbf{mod}\, 2 = 0 \Rightarrow$$
$$z + \underline{(2 * x) * (y \,\mathbf{div}\, 2)} = X * Y$$

$\Downarrow$ [commutativity, associativity of $*$]

$$z+x*y = X*Y \land y \neq 0 \land y \,\mathbf{mod}\, 2 = 0 \Rightarrow z+x*(\underline{2 * (y \,\mathbf{div}\, 2)}) = X*Y$$

$\Downarrow$ [$y \,\mathbf{mod}\, 2 = 0 \Rightarrow 2 * (y \,\mathbf{div}\, 2) = y$]

$$\underline{z + x * y = X * Y} \land y \neq 0 \land y \,\mathbf{mod}\, 2 = 0 \Rightarrow \underline{z + x * y = X * Y}$$

$\Downarrow$ [goal is in the premises]

$$true$$

Program Verification
Table of Examples

PROGRAM	COMPUTING	VARS	LOOPS	INVARIANT GENERATION	VERIFICATION
cohencu	cube	4	1	2.15	0.61
prodbin	product	5	1	2.27	0.44
prod4br	product	6	1	3.32	0.82
hard	integer division	6	2	7.43	5.46
divbin	integer division	5	2	4.28	1.99
sqrt	integer sqrt	4	1	2.26	0.65
dijkstra	integer sqrt	5	2	4.73	18.88
euclidex2	Bezout's coefs	8	1	3.64	1.79
lcm2	lcm	6	1	2.02	0.90
fermat2	divisor	5	1	2.26	1.01
fermatorig2	divisor	5	1	1.88	0.77
factor	divisor	6	1	24.56	2.03

Plan of the Talk

1. Introduction
2. Invariant Generation
3. Program Verification
4. **Future Work and Conclusions**

Future Work

- Verification condition generator: **new data structures**
  - Arrays
  - Lists/stacks/queues
  - Pointers
- Theorem prover: **domain-specific theorem provers**
  - CVC Lite
  - Simplify
  - ICS
  - Decision procedures in our research group
- Loop invariant generator:
  specialized for data structures of interest

Conclusions

- General technique for discovering invariants in loops without nesting
- Shown conditions to compute the *strongest* possible invariant expressible in the assertion language
- Instantiated for finding invariant polynomial equalities
- Applied to program verification with prototype verifier