Loop Invariants

Verification

- Consider a loop-free program P
  - With conditionals
  - Memory references
  - Data structures
  - No function calls
- What is the computational complexity of verifying

  { Precondition } P { Postcondition}

Loops

- Now consider the same problem
  - Where P can have one loop
  - But still no function calls
- What is the computational complexity of verifying

  { Precondition } P { Postcondition}

Verification of Loops

- Verifying properties of loops is *the* hard problem
- Solve this, and everything else is much easier

A Simple Example

```
X = 0
I = 0
while I < 10 do
      X = X + 1
      I = I + 1

assert(X == 10)
```


Loop Invariants

- To verify loops, it suffices to find a sufficiently strong loop invariant
- What is a loop invariant?
  - A predicate that holds on every loop iteration
  - (at the same program point, usually at loop head)
- What is "sufficiently strong"
  - More in a minute …

Loop Invariant (1)

```
X = 0
I = 0
while I < 10 do
      { true }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Loop Invariant (2)

```
Z = 42
X = 0
I = 0
while I < 10 do
      { Z = 42 }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Loop Invariant (3)

```
Z = 42
X = 0
I = 0
while I < 10 do
      { I < 4327 }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Loop Invariant (4)

```
Z = 42
X = 0
I = 0
while I < 10 do
      { X < 11 }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Loop Invariant (5)

```
Z = 42
X = 0
I = 0
while I < 10 do
      { X = I && I < 11 }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Comments

- Loop invariants aren't hard to compute
  - If you don't care about quality
  - true
- What we want is to prove the assertion
  - Need an invariant strong enough to do this

Comments

- But how can we prove the assertion?
- We need a proof strategy
  - An algorithm that we can apply to *any* loop

Inductive Invariants

```
            Pre
while (B)   I
{
 … code …
}           Post
```

$Pre \Rightarrow I$

$I \wedge B$
{ code }
I

$I \wedge \neg B \Rightarrow$
Post

Inductive Invariants

- $Pre \Rightarrow I$
  The invariant holds initially
- $I \wedge B$ { code } I
  If the invariant and loop condition hold, executing the loop body re-establishes the invariant
- $I \wedge \neg B \Rightarrow Post$
  If the invariant holds and the loop terminates, then the post-condition holds

Loop Invariant (1)

```
X = 0
I = 0
while I < 10 do
    { true }
    X = X + 1
    I = I + 1

assert(X == 10)
```


Loop Invariant (2)

```
Z = 42
X = 0
I = 0
while I < 10 do
    { Z = 42 }
    X = X + 1
    I = I + 1

assert(X == 10)
```


Loop Invariant (3)

```
Z = 42
X = 0
I = 0
while I < 10 do
    { I < 4327 }
    X = X + 1
    I = I + 1

assert(X == 10)
```

Loop Invariant (4)

```
Z = 42
X = 0
I = 0
while I < 10 do
      { X < 11 }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Loop Invariant (5)

```
Z = 42
X = 0
I = 0
while I < 10 do
      { X = I && I < 11 }
      X = X + 1
      I = I + 1

assert(X == 10)
```


Invariant Inference

- An old problem

- A different approach with two ideas:
  1. Separate invariant inference from the rest of the verification problem

Invariant Inference

- An old problem

- Two ideas:
  1. Separate invariant inference from the rest of the verification problem

  2. Guess the invariant from executions

Why?

- Complementary to static analysis
  - underapproximations
  - "see through" hard analysis problems
    - functionality may be simpler than the code

- Possible to generate many, many tests

Nothing New Under the Sun

- Sounds like DAIKON?
  - Yes!

- Hypothesize (many) invariants
  - Run the program
  - Discard candidate invariants that are falsified
  - Attempt to verify the remaining candidates

A Simple Program

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Instrument loop head
- Collect state of program variables on each iteration

A DAIKON-Like Approach

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Hypothesize
  - s = y
  - s = 2y
- Data

s	y
0	0

A DAIKON-Like Approach

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Hypothesize
  - s = y
  - ~~s = 2y~~
- Data

s	y
0	0
1	1

A DAIKON-Like Approach

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Hypothesize
  - s = y
  - ~~s = 2y~~
- Data

s	y
0	0
1	1
2	2
3	3

Another Approach

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + 1;
  y := y + 1;
}
```

- Data

s	y
0	0
1	1
2	2
3	3

Arbitrary Linear Invariant

as + by = 0

- Data

s	y
0	0
1	1
2	2
3	3

Observation

as + by = 0

s	y
0	0
1	1
2	2
3	3

w
a
b

=

0
0

Observation

as + by = 0

{ w | Mw = 0 }

s	y
0	0
1	1
2	2
3	3

w
a
b

=

0
0

Observation

as + by = 0

NullSpace(M)

s	y
0	0
1	1
2	2
3	3

w
a
b

=

0
0

Linear Invariants

- Construct matrix M of observations of all program variables
- Compute NullSpace(M)
- All invariants are in the null space

Spurious "Invariants"

- All invariants are in the null space
  - But not all vectors in the null space are invariants
- Consider the matrix

s	y
0	0

- Need a check phase
  - Verify the candidate is in fact an invariant

An Algorithm

- Check candidate invariant
  - If an invariant, done
  - If not an invariant, get counterexample
    - Counterexample can be guaranteed to satisfy all invariants
- Add new row to matrix
  - And repeat

Termination

- How many times can the solve & verify loop repeat?
- Each counterexample is linearly independent of previous entries in the matrix
- So at most N iterations
  - Where N is the number of columns
  - Upper bound on steps to reach a full rank matrix

Summary

- Superset of all linear invariants can be obtained by a standard matrix calculation
- Counter-example driven improvements to eliminate all but the true invariants
  - Guaranteed to terminate

What About Non-Linear Invariants?

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + y;
  y := y + 1;
}
```


Idea

- Collect data as before
- But add more columns to the matrix
  - For derived quantities
  - For example, y^2 and s^2
- How to limit the number of columns?
  - All monomials up to a chosen degree d

[Nguyen, Kapur, Weimer, Forrest 2012]

What About Non-Linear Invariants?

```
s = 0;
y = 0;
while( * )
{
  print(s,y);
  s := s + y;
  y := y + 1;
}
```

1	s	y	s^2	y^2	sy
1	0	0	0	0	0
1	1	1	1	1	1
1	3	2	9	4	6
1	6	3	36	9	18
1	10	4	100	16	40

Solve for the Null Space

$a + bs + cy + ds^2 + ey^2 + fsy = 0$

1	s	y	s^2	y^2	sy
1	0	0	0	0	0
1	1	1	1	1	1
1	3	2	9	4	6
1	6	3	36	9	18
1	10	4	100	16	40

w
a
b
c
d
e
f

=

0
0
0
0
0
0

Candidate invariant: $-2s + y + y^2 = 0$

Comments

- Same issues as before
  - Must check candidate is implied by precondition, is inductive, and implies the postcondition on termination
  - Termination of invariant inference guaranteed if the verifier can generate counterexamples
- Experience: Solvers do well as checkers!

Experiments

Name	#vars	deg	Data	#and	Guess time (sec)	Check time (sec)	Total time (sec)
Mul2	4	2	75	1	0.0007	0.010	0.0107
LCM/GCD	6	2	329	1	0.004	0.012	0.016
Div	6	2	343	3	0.454	0.134	0.588
Bezout	8	2	362	5	0.765	0.149	0.914
Factor	5	3	100	1	0.002	0.010	0.012
Prod	5	2	84	1	0.0007	0.011	0.0117
Petter	2	6	10	1	0.0003	0.012	0.0123
Dijkstra	6	2	362	1	0.003	0.015	0.018
Cubes	4	3	31	10	0.014	0.062	0.076
geoReihe1	3	2	25	1	0.0003	0.010	0.0103
geoReihe2	3	2	25	1	0.0004	0.017	0.0174
geoReihe3	4	3	125	1	0.001	0.010	0.011
potSumm1	2	1	5	1	0.0002	0.011	0.0112
potSumm2	2	2	5	1	0.0002	0.009	0.0092
potSumm3	2	3	5	1	0.0002	0.012	0.0122
potSumm4	2	4	10	1	0.0002	0.010	0.0102

Summary to This Point

- Algorithm for algebraic invariants
  - Up to a given degree
- Guess and Check
  - Hard part is inference done by matrix solve
  - Check part done by standard SMT solver
  - Much simpler and faster than previous approaches

What About Disjunctive Invariants?

- Disjunctions are expensive
  - In addition to conjunctions
- Existing techniques severely restrict disjunctions
  - E.g., to a template

What About Non-Numeric Invariants?

- Arrays?
- Lists?
- Other data structures?
- Invariant inference techniques are very specialized

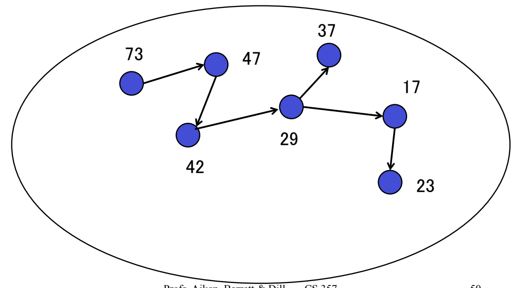
A Search-Based Approach

- All methods for finding invariants are heuristics
  - Can never be complete
- So why not use general but incomplete techniques?

MCMC

- Markov Chain Monte Carlo sampling
- The only known tractable solution method for high dimensional irregular search spaces

MCMC Overview



MCMC Sampling Algorithm for Invariants

1. Select an initial candidate

2. Repeat (millions of times)
   - Propose a random modification and evaluate cost
   - If (cost decreased)
     { accept }
   - If (cost increased)
     { with some probability accept anyway }

Recall

$\text{Pre} \Rightarrow I$

$I(s) \Rightarrow I(t)$ if s {body} t

$I \wedge \neg B \Rightarrow \text{Post}$

Data

- Good states *G*
  - Reachable states
- Pairs *Z*
  - States (s,t) such that starting the loop body S in state s terminates in state t.
- Bad states *B*
  - States that lead to an assertion violation

Cost Function (Roughly)

- Penalize a candidate invariant *C*
  - 1 for each good state g in *G* where *C*(g) is false.
  - 1 for each bad state b in B where C(b) is true
  - 1 for each pair (s,t) in Z where C(s) and not C(t)
- The cost of *C* is the sum of the penalties

Overall Algorithm

- Run search until a 0-cost candidate *C* is found
- Use a decision procedure to verify that *C* is an invariant
  - If yes, done
  - If no, get a counterexample
    - A good state, bad state, or pair
    - Add to the data
    - Repeat

MCMC Sampling Algorithm for Invariants

1. Select an initial candidate

2. Repeat (millions of times)
   - Propose a random modification and evaluate cost
   - If (cost decreased)
     { accept }
   - If (cost increased)
     { with some probability accept anyway }

Numerical Invariants

- Find invariants of the form

$$\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \sum_{k=1}^{n} w_k^{(i,j)} x_k \le d^{(i,j)}$$

Moves

- Replace a coefficient
- Replace a constant on the rhs
- Replace all coefficients and the constant in a single inequality

$$\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \sum_{k=1}^{n} w_k^{(i,j)} x_k \le d^{(i,j)}$$

Results

Program	Z3-H	ICE	[50]	[30]	Pure	MCMC	Templ
cgr1 [27]	0.02	0.2	0.2	0.1	0.05	0.03	0.02
cgr2 [27]	0.03	2.1	?	?	0.68	1.49	1.17
ex7 [33]	0.02	1.1	0.4	?	0.08	0.05	0.04
ex11 [3]	0.03	0.5	0.2	0.1	0.04	0.03	0.05
ex14 [33]	0.01	0.2	0.2	?	0.05	0.03	0.02
ex23 [33]	?	7.3	?	?	0.16	0.13	0.11
fig1 [27]	0.02	1.0	?	?	4.42	0.95	1.44
fig3 [24]	0.01	0.5	0.1	0.1	0.23	0.04	0.04
fig9 [24]	0.02	0.9	0.2	0.1	0.01	0.02	0.01
monniaux	5.14	0.1	1.0	0.2	0.05	0.01	0.03
nested	0.02	?	1.0	0.04	5.21	0.29	2.12
tacas [34]	TO	4.8	0.5	0.1	0.75	0.52	0.08
w1 [27]	0.02	0.5	0.2	0.1	0.05	0.01	0.02
w2 [27]	0.02	0.4	0.1	0.1	0.09	0.03	0.05
array [3]	0.03	1.3	0.2	?	0.24	0.22	0.29
fil1 [3]	0.01	0.2	0.3	0.4	0.01	0.01	0.01
trex01 [3]	0.01	0.2	0.4	0.1	0.03	0.01	0.03

Arrays

- Use the fluid updates abstraction
- Reduce to search for numerical predicate T
  - But now involves universal quantifier
  - f,g are array variables

$$\forall u, v. T(x_1, x_2, \ldots, x_n, u, v) \Rightarrow f[u] = g[v]$$

A Problem with Arrays

- Decision procedures for arrays cannot give us counterexamples
- Instead use executions to generate data
  - Including bad states and pairs

Generating Data

- Pick a number k
- At the loop head
  - Assign all numeric variables a value <= k
  - Assign all arrays a size <= k
  - Assign all elements of arrays a value <= k
- For experiments, we used k = 4

Results on Arrays

Program	[17]	Z3-H	ARMC	Dual	Pure	MCMC	Templ
init	0.01	0.06	0.15	0.72	0.06	0.02	0.01
init-nc	0.02	0.08	0.48	6.60	0.05	0.15	0.02
init-p	0.01	0.03	0.14	2.60	0.01	0.01	0.01
init-e	0.04	TO	TO	TO	TO	TO	TO
2darray	0.04	0.18	?	TO	0.02	0.41	0.02
copy	0.01	0.04	0.20	1.40	0.87	0.80	0.02
copy-p	0.01	0.04	0.21	1.80	0.09	0.13	0.01
copy-o	0.04	TO	?	4.50	TO	TO	0.50
reverse	0.03	0.12	2.28	8.50	TO	3.48	0.03
swap	0.12	0.41	3.0	40.60	TO	TO	0.21
d-swap	0.16	1.37	4.4	TO	TO	TO	0.51
strcpy	0.07	0.05	0.15	0.62	0.01	0.02	0.01
strlen	0.02	0.07	0.02	0.20	0.01	0.01	0.01
memcpy	0.04	0.20	16.30	0.20	0.02	0.03	0.01
find	0.02	0.01	0.08	0.38	2.23	0.30	0.02
find-n	0.02	0.01	0.08	0.39	0.07	0.95	0.01
append	0.02	0.04	1.76	1.50	TO	TO	0.12
merge	0.09	0.04	?	1.50	TO	TO	0.41
alloc-f	0.02	0.02	0.09	0.69	0.07	0.10	0.01
alloc-nf	0.03	0.03	0.13	0.42	0.49	0.14	0.07

Strings

- Search space is
  - Boolean combinations of predicates P
  - P consists of constants and predicates in the program

```
i:=0; x:="a";
while(non_det()){i++; x:= "("+x+")"}
assert( x.length == 2*i+1);
if(i>0) assert(x.contains("(a)");
```


String Results

	Figure 2	replace	index	substring
Pure	342.59	0.01	0.06	0.53
MCMC	0.82	0.02	0.06	0.05
Z3-STR	0.03	TO	114.58	0.01

Lists

- Search space
  - Boolean combinations of atoms
  - Atoms are relations $R(x_1,\ldots,x_n)$
- Moves
  - Replace one argument of a relation
  - Replace an entire relation
  - Flip polarity of an atom

Lists

- Use one reachability relation

n(x,y) = y is reachable from x in 0 or more pointer dereferences

List Results

Program	#G	#R	Search	Valid.	Prop.	Accep.
delete	50	2	0.20	0.04	4437	3949
delete-all	20	7	1.03	0.13	8482	7225
find	50	9	0.42	0.04	6681	5560
filter	50	26	10.41	0.11	160489	126389
last	50	3	0.90	0.04	98064	87446
reverse	20	54	55.11	0.08	582665	484208

Summary

- Invariant inference is a hard problem, made easier by looking at data from executions
  - Because the executions satisfy all the invariants
- Search-based techniques can work
  - Competitive with other methods
  - Easier to retarget to new domains
- Still limited by decision procedures
  - But not by their ability to do inference