ALICe: A Benchmark to Improve Affine Loop Invariant Computation

Vivien Maisonneuve

MINES ParisTech

Seventh meeting of the French community of compilation

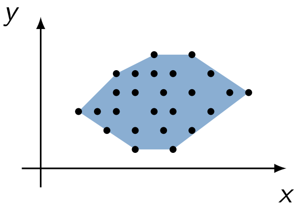
Dammarie-les-Lys, December 2013

Introduction

Program analysis $\Rightarrow$ computation of invariants (e.g. model checking).

Need of abstract domains to represent complex program behaviors.

Here: affine invariants = systems of linear (in)equations.



Linear Relation Analysis

Predicate propagation: forward / backward.

```
x = 0; y = 0;
while (x <= 100) {
  b = rand();
  if (b) x += 2;
  else x += 1, y += 1;

}
```

Linear Relation Analysis

Predicate propagation: forward / backward.

```
x = 0; y = 0;  // x = y = 0
while (x <= 100) {
  b = rand();
  if (b) x += 2;
  else x += 1, y += 1;

}
```

Linear Relation Analysis

Predicate propagation: forward / backward.

```
x = 0; y = 0;  // x = y = 0
while (x <= 100) {  // x = y = 0
  b = rand();
  if (b) x += 2;
  else x += 1, y += 1;

}
```

Linear Relation Analysis

Predicate propagation: forward / backward.

```
x = 0; y = 0;  // x = y = 0
while (x <= 100) {  // x = y = 0
  b = rand();
  if (b) x += 2;  // x = 2, y = 0
  else x += 1, y += 1;  // x = 1, y = 1

}
```

Linear Relation Analysis

Predicate propagation: forward / backward.

Branches: convex union of invariants.

```
x = 0; y = 0;  // x = y = 0
while (x <= 100) {  // x = y = 0
  b = rand();
  if (b) x += 2;  // x = 2, y = 0
  else x += 1, y += 1;  // x = 1, y = 1
  // 1 ≤ x ≤ 2, x + y = 2
}
```

Linear Relation Analysis

Predicate propagation: forward / backward.

Branches: convex union of invariants.

Loops? Widening ⇒ // $0 \le y \le x$

```
x = 0; y = 0;  // x = y = 0
while (x <= 100) {  // x = y = 0
  b = rand();
  if (b) x += 2;  // x = 2, y = 0
  else x += 1, y += 1;  // x = 1, y = 1
  // 1 ≤ x ≤ 2, x + y = 2
}
```

Linear Relation Analysis

Predicate propagation: forward / backward.

Branches: convex union of invariants.

Loops? Widening ⇒ // $0 \leq y \leq x$

```
x = 0; y = 0;  // x = y = 0
while (x <= 100) {  // x = y = 0
  b = rand();
  if (b) x += 2;  // x = 2, y = 0
  else x += 1, y += 1;  // x = 1, y = 1
  // 1 ≤ x ≤ 2, x + y = 2
}
```

Sources of approximation:

- Branches ⇒ convex hull
- Loops ⇒ lots of research, programs

ALICe

Benchmark to compare several techniques & programs to compute affine loop invariants.
http://alice.cri.mines-paristech.fr/

Motivations:

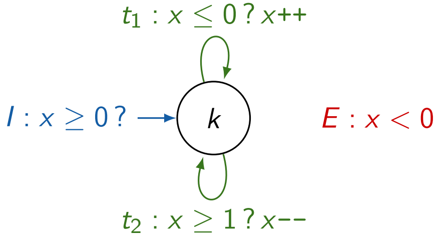
1. Compare tools on a common set of previously published examples.
2. Study effects of input model restructurations.
3. Improve invariant computation in PIPS.

Contents

Models

Transition systems with a finite number of vertices ("control states"), of integer variables.

- Initial condition I on control states & variables.
- Transitions $t_1, \ldots, t_n$ with guards and actions.
- Error condition E on control states & variables.

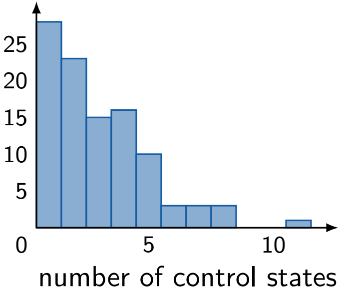


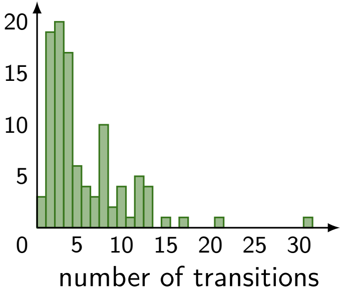
Goal: E is unreachable.

Test Cases

102 previously published test cases: from L. Gonnord, S. Gulwani, N. Halbwachs, B. Jeannet et al.

Small test cases: 1-10 control states, 2-15 transitions.
Mostly: loop invariants, loop bounds, protocols.





Tools

Supported tools:

- **Aspic**
- **isl**
- **PIPS**

Tools

Supported tools:

- **Aspic**: polyhedral invariant generator. Developed by L. Gonnord. Forward LRA + **accelerations**.
- **isl**
- **PIPS**

Tools

Supported tools:

- **Aspic**: polyhedral invariant generator. Developed by L. Gonnord. Forward LRA + **accelerations**.
- **isl**: the Integer Set Library. Developed by S. Verdoolaege. A library for manipulating sets and relations of integer tuples bounded by affine constraints:

  $$S(s) = \{x \in \mathbb{Z}^d \mid \exists z \in \mathbb{Z}^e : Ax + Bs + Dz \geq c\}$$
  $$R(s) = \{x_1 \to x_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists z \in \mathbb{Z}^e : A_1x_1 + A_2x_2 + Bs + Dz \geq c\}$$

  more expressive than polyhedra ($\sim$ Presburger).

  Models as relations.
  Sophisticated computation of transitive closure.
- **PIPS**

PIPS

Interprocedural source-to-source compiler framework for C and Fortran.
Initially developed at MINES ParisTech.

Code analysis: 2-step approach

1. Program is abstracted: each program command instruction (elementary or compound) is associated to an affine transformer that represents the transfer function.
   Bottom-up procedure.

```
while (rand())
  x += 2;          //
```
$T = \{(x, x') \mid x' = x + 2\}$

Notation: x before, x' after.

PIPS

Interprocedural source-to-source compiler framework for C and Fortran.
Initially developed at MINES ParisTech.

Code analysis: 2-step approach

1. Program is abstracted: each program command instruction (elementary or compound) is associated to an affine transformer that represents the transfer function.
   Bottom-up procedure.

```
while (rand())   //   T* = {(x, x') | x' ≥ x}
  x += 2;        //   T = {(x, x') | x' = x + 2}
```

Notation: x before, x' after.

PIPS

Interprocedural source-to-source compiler framework for C and Fortran.
Initially developed at MINES ParisTech.

Code analysis: 2-step approach

1. Program is abstracted: each program command instruction (elementary or compound) is associated to an affine transformer that represents the transfer function.
   Bottom-up procedure.
2. Then, invariants

```
//   P = {x | 0 ≤ x ≤ 42}
while (rand())  //   T* = {(x, x') | x' ≥ x}
  x += 2;       //   T = {(x, x') | x' = x + 2}
```

Notation: x before, x' after.

PIPS

Interprocedural source-to-source compiler framework for C and Fortran.
Initially developed at MINES ParisTech.

Code analysis: 2-step approach

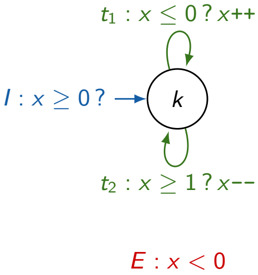
1. Program is abstracted: each program command instruction (elementary or compound) is associated to an affine transformer that represents the transfer function.
   Bottom-up procedure.
2. Then, invariants are propagated along transformers.

```
//   P = {x | 0 ≤ x ≤ 42}
while (rand())   //   T* = {(x, x') | x' ≥ x}
  x += 2;        //   T = {(x, x') | x' = x + 2}
//   P' = {x | 0 ≤ x}
```

Notation: x before, x' after.

Input Format

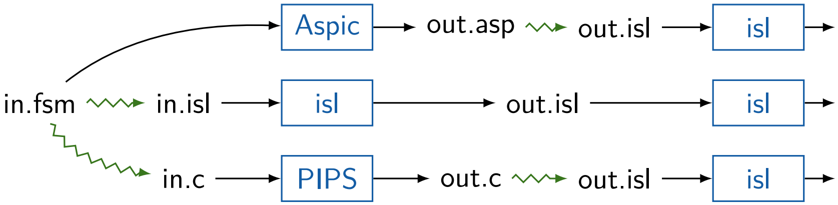
Test cases are written in fsm format (Aspic format, introduced by FAST).




```
model M {
  var x;
  states k;
  transition t1 {
    from := k;
    to := k;
    guard := x <= 0;
    action := x' = x + 1;
  }
  transition t2 {
    ...
  }
}
strategy S {
  Region init := {x >= 0};
  Region bad := {x < 0};
}
```

Easy, existing base of models, c2fsm.

Test Chain



To challenge a tool T on a test case:

- convert test case into T's input language.
- run T, get the resulting invariant in T's output language;
- convert invariant in isl format;
- check with isl that the invariant does not reach the error region.

$\Rightarrow$ Several wrappers and format conversion tools involved.

Mostly written in OCaml, wrappers in Python.

Comparative Results

Out of 102 test cases:

	Aspic	isl	PIPS
Successes	75	63	43
Time (s.)	10.9	35.5	46.2

(Quad-core AMD Opteron Processor 2380 at 2.4 GHz, 16 GB RAM)

Remarks:

- Best results with Aspic (native format, ad-hoc tool).
- isl very good with loops, not at ease with multiple states.
  Very fast on small cases, slower on bigger ones.
- Average results with PIPS (default options).
  Slower, poor results with parallel loops.

Contents

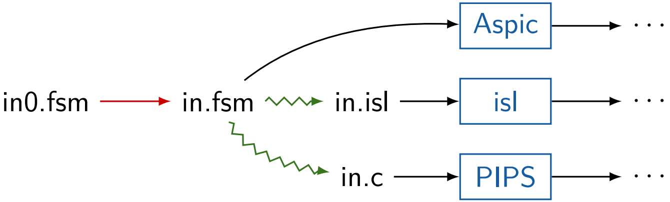
Model Restructurations

A strategy to improve results: restructure the input model into an equivalent one, easier to analyze.

Formally, a model transformation is a function: $M_1 \longmapsto M_2$ s.t.

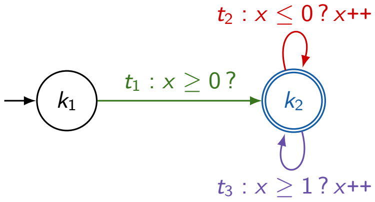
$$M_2 \text{ correct (unreachable error region)} \Longrightarrow M_1 \text{ correct.}$$

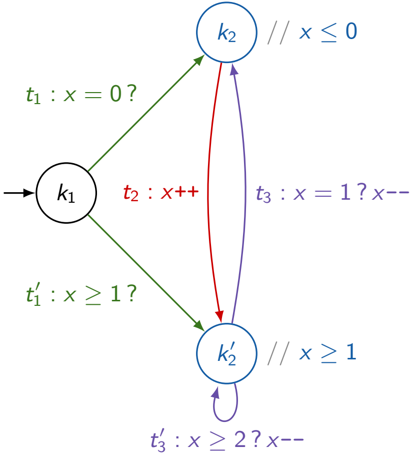
Implemented in ALICe: source-to-source fsm transformation before analysis.



State Splitting Heuristic

Designed to improve results in PIPS: get rid of nodes with several self loops that PIPS has difficulty to analyze [NSAD'11].
Nodes split according to the guards of the loops.

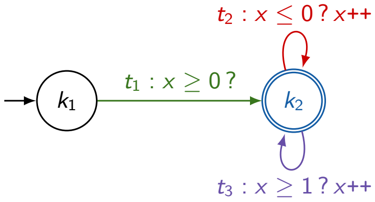


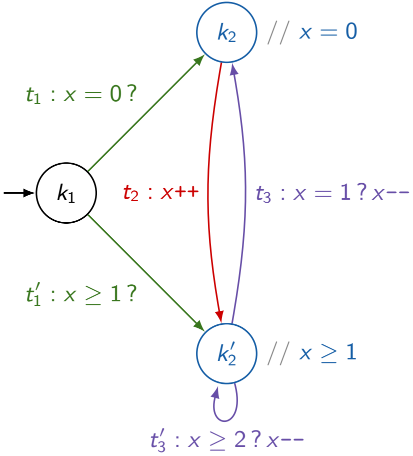


Proved in Coq.

State Splitting Heuristic

Designed to improve results in PIPS: get rid of nodes with several self loops that PIPS has difficulty to analyze [NSAD'11].
Nodes split according to the guards of the loops.



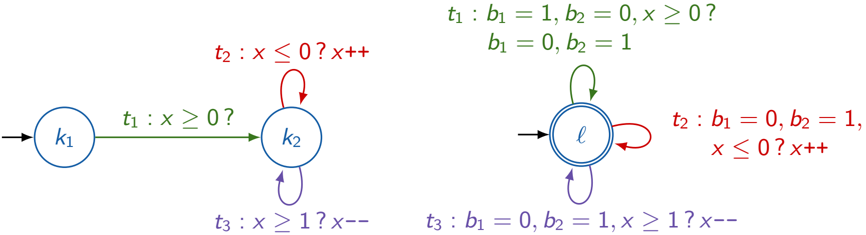


Proved in Coq.

Using an Unique State

Transformation to recode the model s.t. it contains only one node ℓ:

- all transitions turned into loops on ℓ;
- extra variables $b_i = 1$ if in state k_i of the original model, 0 otherwise.



Purposes:

- produce more stressful test cases;
- test isl behavior;
- reduce bias factors related to encoding choices;
- can be used prior the state splitting heuristic, increasing its effects.

Results

Out of 102 test cases:

	Aspic	isl	PIPS
Direct			
Successes	75	63	43
Time (s.)	10.9	35.5	46.2
Split			
Successes	79	72	50
Time (s.)	12.8	43.0	61.7
Merged			
Successes	59	70	40
Time (s.)	16.7	26.2	50.0
Merged + Split			
Successes	70	83	63
Time (s.)	11.3	40.8	59.5

Remarks:

- Splitting helps all tools.
- Merging helps isl: very good with loops, not at ease with multiple states in direct encoding.
- Best results obtained through merging + splitting, except for Aspic: unaccelerable transitions.
- Slowdown in most cases: more complicated structure.

Improving Results in PIPS

Several options in PIPS to improve analysis results.

1. **Delay Convex Hulls** at step 1, using transformer lists
   Consider a loop with 2 control paths defined by transformers T_1, T_2, and precondition P.
   By default, loop body is abstracted by a unique transformer so postcondition P' is: $P' = (T_1 \sqcup T_2)^*(P)$, inaccurate.
   With TL,
   $$P' = \Big[\mathrm{Id} \sqcup T_1 \sqcup T_2 \sqcup (T_1 \circ T_2) \sqcup (T_2 \circ T_1) \sqcup {T_1}^+ \sqcup {T_2}^+ \sqcup {T_1}^+ \circ T_2 \circ (T_1 \sqcup T_2)^* \sqcup {T_2}^+ \circ T_1 \circ (T_1 \sqcup T_2)^* \Big](P)$$
   Convex hull is delayed, each elementary transition T_i is applied, more information is preserved.
2. **Perform Iterative Analysis**
3. **Use Multiple Precision Arithmetic**

Improving Results in PIPS

Several options in PIPS to improve analysis results.

1. **Delay Convex Hulls**
2. **Perform Iterative Analysis**
   Use preconditions to refine transformers on a second pass:
   - Compute loop transformer $T(\bar{x}, \bar{x}')$.
     Compute loop invariant $P(\bar{x})$, using T.
   - Compute loop transformer $T'(\bar{x}, \bar{x}') = T(\bar{x}, \bar{x}') \wedge P(\bar{x}) \wedge P(\bar{x}')$.
     Compute loop invariant $P'(\bar{x})$, using T'.
3. **Use Multiple Precision Arithmetic**

Improving Results in PIPS

Several options in PIPS to improve analysis results.

1. **Delay Convex Hulls**
2. **Perform Iterative Analysis**
3. **Use Multiple Precision Arithmetic**
   If intermediate computations raise polyhedrons with huge coefficients: arithmetic error, constraint loss.
   $\Rightarrow$ GMP.

Options can be combined.

Results for PIPS Options

Out of 102 test cases:

Options	None	TL	IA	TL + IA	TL + IA + MP
Direct					
Successes	43	69	45	72	73
Time (s.)	46.2	51.4	69.3	74.8	113.2
Split					
Successes	50	72	56	75	77
Time (s.)	61.7	68.9	93.5	102.5	156.3
Merged					
Successes	40	66	44	67	68
Time (s.)	50.0	55.8	75.3	82.5	126.6
Merged + Split					
Successes	63	79	65	80	82
Time (s.)	59.5	66.6	90.2	98.5	146.3

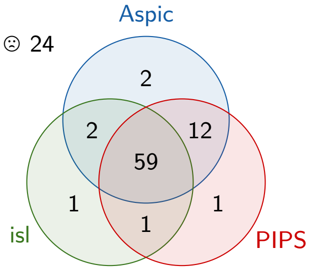
Combine options and/or restructurations.

Comparative Results, Revisited

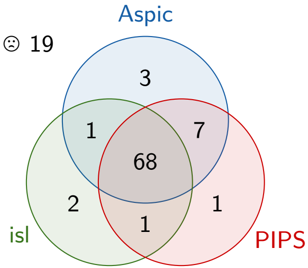
Out of 102 test cases:

	Aspic	isl	PIPS default	PIPS + options
Direct				
Successes	75	63	43	73
Time (s.)	10.9	35.5	46.2	113.2
Split				
Successes	79	72	50	77
Time (s.)	12.8	43.0	61.7	156.3
Merged				
Successes	59	70	40	68
Time (s.)	16.7	26.2	50.0	126.6
Merged + Split				
Successes	70	83	63	82
Time (s.)	11.3	40.8	59.5	146.3

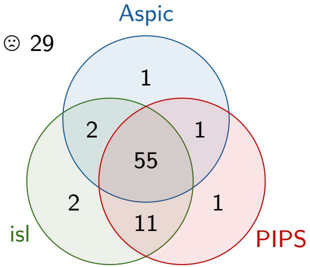
Comparative Results



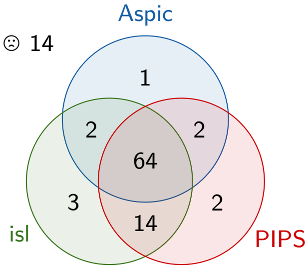
Direct



Split



Merge



Merge + Split

Conclusion

What has been done

- Collection of test cases.
- Working with 3 tools: Aspic, isl, PIPS, handling various formats.
- Restructurations.

Future work

- Study failures: by tool, by type. Find patterns?
- FASTer backed.
- Improve restructurations.
- Avoid cheating: minimal invariant?

ALICe: A Benchmark to Improve Affine Loop Invariant Computation

Vivien Maisonneuve

MINES ParisTech

Seventh meeting of the French community of compilation

Dammarie-les-Lys, December 2013