ALICe: A Framework to Improve Affine Loop Invariant Computation

Vivien Maisonneuve

Olivier Hermant  François Irigoin

5th International Workshop on Invariant Generation (WING)
July 23, 2014
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.

```c
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.

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

Sources of approximation:

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

\[ \begin{align*}
  x &= 0; \\ y &= 0; & // & x = y = 0 \\
  \text{while} & \ (x \leq 100) \{ & // & x = y = 0 \\
     & b = \text{rand}(); \\
     & \text{if} \ (b) \ x += 2; & // & x = 2, y = 0 \\
     & \text{else} \ x += 1, y += 1; & // & x = 1, y = 1 \\
  \}
\]
Linear Relation Analysis

Predicate propagation: forward / backward.

Branches: convex union of invariants.

```plaintext
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 ≤ y ≤ x

```plaintext
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
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; // $x = 1, y = 1$
  // $1 \leq x \leq 2, x + y = 2$
}

Sources of approximation:
- Branches ⇒ convex hull
- Loops ⇒ lots of research, programs
ALICe

Framework to compare several techniques & programs to compute affine loop invariants.

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

1 The Test Suite
   Test Cases
   Supported Tools
   Test Chain
   Results

2 Model Restructurations
   State Splitting Heuristic
   Using a Unique State
   Comparative Results

3 Improving Results in PIPS
   Transformer Lists
   Iterative Analysis
   Multiple Precision Arithmetic
   Results
Test Cases

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 updates.
- **Error condition** \( E \) on control states & variables.

\[
\begin{align*}
t_1 : & \ x \leq 0 \quad \text{?} \quad x++ \\
I : & \ x \geq 0 \quad \text{?} \\
t_2 : & \ x \geq 1 \quad \text{?} \quad x-- \\
E : & \ x < 0
\end{align*}
\]

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.
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;
        update := x' = x + 1;
    }
    transition t2 {
        ...
    }
}
strategy S {
    Region init := {x >= 0};
    Region bad := {x < 0};
}
```

Easy, existing base of models, c2fsm.
Tools

Supported tools:

- Aspic
- isl
- PIPS
Tools

Supported tools:

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

- **PIPS**
Tools

Supported tools:

- **Aspic**: affine 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:

  \[
  S(s) = \{ x \in \mathbb{Z}^d \mid \exists z \in \mathbb{Z}^e : Ax + Bs + Dz \geq c \}
  \]

  \[
  R(s) = \{ x_1 \rightarrow x_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \mid \exists z \in \mathbb{Z}^e : A_1x_1 + A_2x_2 + Bs + Dz \geq c \}
  \]

  more expressive than affine constraints (\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; // $T = \{(x, x') | x' = x + 2\}$

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.

   ```c
   while (rand())  //   \( T^* = \{ (x, x') | x' \geq x \} \)
   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

```plaintext
// P = \{x | 0 \leq x \leq 42\}
while (rand()) // T^* = \{(x, x') | x' \geq 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.

// $P = \{ x | 0 \leq x \leq 42 \}$

while (rand()) // $T^* = \{(x, x') | x' \geq x\}$
  $x += 2$; // $T = \{(x, x') | x' = x + 2\}$

// $P' = \{ x | 0 \leq x \}$

Notation: $x$ before, $x'$ after.
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:

<table>
<thead>
<tr>
<th></th>
<th>Aspic</th>
<th>isl</th>
<th>PIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Successes</strong></td>
<td>75</td>
<td>63</td>
<td>43</td>
</tr>
<tr>
<td><strong>Time (s.)</strong></td>
<td>10.9</td>
<td>35.5</td>
<td>46.2</td>
</tr>
</tbody>
</table>

(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 concurrent loops.
Contents

1 The Test Suite
   Test Cases
   Supported Tools
   Test Chain
   Results

2 Model Restructurations
   State Splitting Heuristic
   Using a Unique State
   Comparative Results

3 Improving Results in PIPS
   Transformer Lists
   Iterative Analysis
   Multiple Precision Arithmetic
   Results
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: \( M_1 \mapsto M_2 \) s.t.

\[ M_2 \text{ correct (unreachable error region)} \implies M_1 \text{ correct}. \]

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 a Unique State

Transformation to recode the model s.t. it contains only one state $\ell$:
- all transitions turned into loops on $\ell$;
- extra variables $b_i = 1$ if in state $k_i$ of the original model, 0 otherwise.

![Diagram showing the transition and loop conditions](image)

**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:

<table>
<thead>
<tr>
<th></th>
<th>Aspic</th>
<th>isl</th>
<th>PIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>75</td>
<td>63</td>
<td>43</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>10.9</td>
<td>35.5</td>
<td>46.2</td>
</tr>
<tr>
<td><strong>Split</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>79</td>
<td>72</td>
<td>50</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>12.8</td>
<td>43.0</td>
<td>61.7</td>
</tr>
<tr>
<td><strong>Merged</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>59</td>
<td>70</td>
<td>40</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>16.7</td>
<td>26.2</td>
<td>50.0</td>
</tr>
<tr>
<td><strong>Merged + Split</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>70</td>
<td>83</td>
<td>63</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>11.3</td>
<td>40.8</td>
<td>59.5</td>
</tr>
</tbody>
</table>

#### 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,
   $$P' = \left[ \text{Id} \sqcup T_1 \sqcup T_2 \sqcup (T_1 \circ T_2) \sqcup (T_2 \circ T_1) \sqcup T_1^+ \sqcup T_2^+ \sqcup T_1^+ \circ T_2 \circ (T_1 \sqcup T_2)^* \sqcup T_2^+ \circ T_1 \circ (T_1 \sqcup T_2)^* \right] (P)$$
   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 \( T'(\bar{x}, \bar{x}') = T(\bar{x}, \bar{x}') \land P(\bar{x}) \land P(\bar{x}') \).
     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 constraints with huge coefficients: arithmetic error, constraint loss.
   - ⇒ GMP.

Options can be combined.
### Results for PIPS Options

Out of 102 test cases:

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

Combine options and/or restructurizations.
## Comparative Results, Revisited

Out of 102 test cases:

<table>
<thead>
<tr>
<th></th>
<th>Aspic</th>
<th>isl</th>
<th>PIPS default</th>
<th>PIPS + options</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Direct</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>75</td>
<td>63</td>
<td>43</td>
<td>73</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>10.9</td>
<td>35.5</td>
<td>46.2</td>
<td>113.2</td>
</tr>
<tr>
<td><strong>Split</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>79</td>
<td>72</td>
<td>50</td>
<td>77</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>12.8</td>
<td>43.0</td>
<td>61.7</td>
<td>156.3</td>
</tr>
<tr>
<td><strong>Merged</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>59</td>
<td>70</td>
<td>40</td>
<td>68</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>16.7</td>
<td>26.2</td>
<td>50.0</td>
<td>126.6</td>
</tr>
<tr>
<td><strong>Merged + Split</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>70</td>
<td>83</td>
<td>63</td>
<td>82</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>11.3</td>
<td>40.8</td>
<td>59.5</td>
<td>146.3</td>
</tr>
</tbody>
</table>
Comparative Results

Aspic

1. Direct:
   - Aspic: 2
   -Isl: 1
   -PIPS: 1
   -Total: 4

2. Split:
   - Aspic: 3
   -Isl: 1
   -PIPS: 1
   -Total: 5

3. Merge:
   - Aspic: 1
   -Isl: 2
   -PIPS: 1
   -Total: 4

4. Merge + Split:
   - Aspic: 1
   -Isl: 2
   -PIPS: 2
   -Total: 5
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/PAGAI backed.
- Improve restructurations.
- Avoid cheating: minimal invariant?
ALICe: A Framework to Improve Affine Loop Invariant Computation

Vivien Maisonneuve

Olivier Hermant François Irigoin

5th International Workshop on Invariant Generation (WING)
July 23, 2014