Convex Invariant Refinement by Control Node Splitting: a Heuristic Approach

Vivien Maisonneuve

CRI, Mathématiques et systèmes
MINES ParisTech

September 13, 2011

Context

Working with PIPS: "a source-to-source compilation framework for analyzing and transforming C and Fortran programs", initiated by MINES ParisTech.

Used for program analysis.

Most of program analysis techniques consist in starting from a set of supposed predicates about a particular position in the transition system, and then propagating it to other positions by evaluating the effect of each transition on the predicates.

Particularity of PIPS: computes state transformers = transfer functions, before state predicates.

Context

Working with PIPS: "a source-to-source compilation framework for analyzing and transforming C and Fortran programs", initiated by MINES ParisTech.

Used for program analysis.

Most of program analysis techniques consist in starting from a set of supposed predicates about a particular position in the transition system, and then propagating it to other positions by evaluating the effect of each transition on the predicates.

Particularity of PIPS: computes state transformers = transfer functions, before state predicates.

Goal: improve the accuracy of invariants found when analyzing a TS.

Transform the program.

Transformer

Let

- Var a finite set of n typed variables.
- Val the set of valuations on Var.

A transformer T is a relation from Val to Val: $T \subseteq \text{Val} \times \text{Val}$.

Transformer

Let

- Var a finite set of n typed variables.
- Val the set of valuations on Var.

A transformer T is a relation from Val to Val: $T \subseteq \text{Val} \times \text{Val}$.

T (over)approximates the behavior of a piece of code c if, for all valuations $v, v' \in \text{Val}$:

$$\begin{gathered} c \text{ called on vars. equal to } v \text{ may result in vars. equal to } v' \\ \Downarrow \\ (v, v') \in T \end{gathered}$$

Example

Let x an integer variable, the instruction

```
x += 2;
```

is represented by the transformer

$$T = \{(n, n+2) \mid n \in \mathbb{Z}\}$$

Affine Transformers

An affine transformer is a transformer whose constraints form a convex polyhedron.

Can also be expressed as a conjunction of affine (in)equalities on $2n$ integer variables $x_1 \dots x_n$ (initial values), $x'_1 \dots x'_n$ (final values).

Affine Transformers

An affine transformer is a transformer whose constraints form a convex polyhedron.

Can also be expressed as a conjunction of affine (in)equalities on $2n$ integer variables $x_1 \dots x_n$ (initial values), $x'_1 \dots x'_n$ (final values).

$T = \{(n, n+2) \mid n \in \mathbb{Z}\}$ is an affine transformer, expressible with the affine equality

$$x' = x + 2$$

Affine Transformer Analysis

PIPS approach:

- Affine transformers are used to approximate each program command, elementary or compound statement or procedure call.
- Each function is analyzed once and its transformer is reused at each call site.
- Invariants are propagated using the transformers.

Example 1

Consider a simple program with one variable x.

ℓ_1:
```
x = 0;
```
ℓ_2:
```
while (rand())
```
ℓ_3:
```
    x += 2;
```

Example 1

Consider a simple program with one variable x.

ℓ_1: `x = 0;`

ℓ_2: `while (rand())`

ℓ_3: `x += 2;` // $T_{\ell_3} = \{x' = x + 2\}$

Example 1

Consider a simple program with one variable x.

ℓ_1: `x = 0;` // $T_{\ell_1} = \{x' = 0\}$

ℓ_2: `while (rand())`

ℓ_3: `    x += 2;` // $T_{\ell_3} = \{x' = x + 2\}$

Example 1

Consider a simple program with one variable `x`.

ℓ_1: `x = 0;` // $T_{\ell_1} = \{x' = 0\}$

ℓ_2: `while (rand())` // $T_{\ell_2} = (T_{\ell_3})^* = \{x' \geq 0\}$

ℓ_3: `x += 2;` // $T_{\ell_3} = \{x' = x + 2\}$

T_{ℓ_2} obtained, for example, by Affine Derivative Closure algorithm.

Computation of loops is factor of inaccuracy.

Example 1

Invariants are computed usually from the program entry point, by propagation along the transformers.

// no invariant
ℓ_1: `x = 0;` // $T_{\ell_1} = \{x' = 0\}$
// ???
ℓ_2: `while (rand())` // $T_{\ell_2} = (T_{\ell_3})^* = \{x' \geq x\}$
ℓ_3: `x += 2;` // $T_{\ell_3} = \{x' = x + 2\}$
// ???

Example 1

Invariants are computed usually from the program entry point, by propagation along the transformers.

// no invariant
ℓ_1: `x = 0;` // $T_{\ell_1} = \{x' = 0\}$
// $x = 0$
ℓ_2: `while (rand())` // $T_{\ell_2} = (T_{\ell_3})^* = \{x' \geq x\}$
ℓ_3: `x += 2;` // $T_{\ell_3} = \{x' = x + 2\}$
// ???

Example 1

Invariants are computed usually from the program entry point, by propagation along the transformers.

// no invariant
ℓ_1: `x = 0;` // $T_{\ell_1} = \{x' = 0\}$
// $x = 0$
ℓ_2: `while (rand())` // $T_{\ell_2} = (T_{\ell_3})^* = \{x' \geq x\}$
ℓ_3: `x += 2;` // $T_{\ell_3} = \{x' = x + 2\}$
// $x \geq 0$

Example 2

We consider another example:

ℓ_1: `x = rand();`
ℓ_2: `while (rand())`
ℓ_3: `{`
ℓ_4: `    if (x > 0) x--;`
ℓ_5: `    else if (x <= 0) x++;`
ℓ_6: `}`

Example 2

We consider another example:

ℓ_1: `x = rand();` // $T_{\ell_1} = \{x' \geq 0\}$
ℓ_2: `while (rand())`
ℓ_3: `{`
ℓ_4: `if (x > 0) x--;` // $T_{\ell_4} = \{x > 0 \wedge x' = x - 1\}$
ℓ_5: `else if (x <= 0) x++;` // $T_{\ell_5} = \{x \leq 0 \wedge x' = x + 1\}$
ℓ_6: `}`

Example 2

We consider another example:

```
ℓ1:  x = rand();      // T_ℓ1 = {x' ≥ 0}
ℓ2:  while (rand())      // T_ℓ2 = (T_ℓ3)* = ?
ℓ3:  {     // T_ℓ3 = ?
ℓ4:      if (x > 0) x--;      // T_ℓ4 = {x > 0 ∧ x' = x − 1}
ℓ5:      else if (x <= 0) x++;      // T_ℓ5 = {x ≤ 0 ∧ x' = x + 1}
ℓ6:  }
```

To compute T_{ℓ_2}, T_{ℓ_3} must be known.

Since both if branches may be taken a priori, $T_{\ell_3} \supseteq T_{\ell_4} \cup T_{\ell_5}$.
Also, T_{ℓ_3} must be affine.
$\Rightarrow$ Best approximation is the convex union

$$\begin{aligned} T_{\ell_3} &= T_{\ell_4} \sqcup T_{\ell_5} \\ &= \{x - 1 \leq x' \leq x + 1\} \end{aligned}$$

Yet, inaccurate operation.

Example 2

We consider another example:

ℓ_1: `x = rand();` // $T_{\ell_1} = \{x' \geq 0\}$
ℓ_2: `while (rand())` // $T_{\ell_2} = (T_{\ell_3})^* = ?$
ℓ_3: `{` // $T_{\ell_3} = \{x - 1 \leq x' \leq x + 1\}$
ℓ_4: `if (x > 0) x--;` // $T_{\ell_4} = \{x > 0 \wedge x' = x - 1\}$
ℓ_5: `else if (x <= 0) x++;` // $T_{\ell_5} = \{x \leq 0 \wedge x' = x + 1\}$
ℓ_6: `}`

$T_{\ell_2} = (T_{\ell_3})^* = \{\}$.
There is no constraint in T_{ℓ_2}!

Example 2

During computation of invariants:

// no invariant
ℓ_1: `x = rand();` // $T_{\ell_1} = \{x' \geq 0\}$
// $x \geq 0$
ℓ_2: `while (rand())` // $T_{\ell_2} = \{\}$
ℓ_3: `{` // $T_{\ell_3} = \{x - 1 \leq x' \leq x + 1\}$
ℓ_4: `if (x > 0) x--;` // $T_{\ell_4} = \{x > 0 \wedge x' = x - 1\}$
ℓ_5: `else if (x <= 0) x++;` // $T_{\ell_5} = \{x \leq 0 \wedge x' = x + 1\}$
ℓ_6: `}`

Example 2

During computation of invariants:

// no invariant
ℓ_1: `x = rand();` // $T_{\ell_1} = \{x' \geq 0\}$
// $x \geq 0$
ℓ_2: `while (rand())` // $T_{\ell_2} = \{\}$
ℓ_3: `{` // $T_{\ell_3} = \{x - 1 \leq x' \leq x + 1\}$
ℓ_4: `if (x > 0) x--;` // $T_{\ell_4} = \{x > 0 \wedge x' = x - 1\}$
ℓ_5: `else if (x <= 0) x++;` // $T_{\ell_5} = \{x \leq 0 \wedge x' = x + 1\}$
ℓ_6: `}`
// no invariant ☹

Issue

What happens:

- Inaccuracy in the computation of effects of parallel paths (`if`... `else`), increased by the $(*)$ operation.
- Occurs when there are parallel loops, i.e. `while`... `if` structures.

Issue

What happens:

- Inaccuracy in the computation of effects of parallel paths (`if... else`), increased by the $(*)$ operation.
- Occurs when there are parallel loops, i.e. `while... if` structures.

To adress the issue:

- Refine transformers involved in loops.
- Get information on order in which parallel loops can be performed.
- Decrease the number of parallel loops.

$\Rightarrow$ Program restructurations.

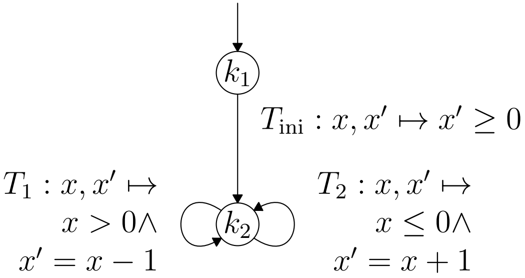
Transformer Automaton

An (affine) transformer automaton is a triplet $\alpha = (K, k_{\text{ini}}, \text{Trans})$ where

- K is a finite set of control points.
- $k_{\text{ini}} \in K$ is the initial control point.
- Trans is a finite set of transitions, i.e. of triplets (k, T, k') with $k, k' \in K$ and T is an (affine) transformer.

Example

```
x = rand();
while (rand())
{
  if (x > 0) x--;
  else if (x <= 0) x++;
}
```



$\alpha = (K, k_{\text{ini}}, \text{Trans})$:

- $K = \{k_1, k_2\}$.
- $k_{\text{ini}} = k_1$.
- $\text{Trans} = \{(k_1, T_{\text{ini}}, k_2), (k_2, T_1, k_2), (k_2, T_2, k_2)\}$.

Semantics

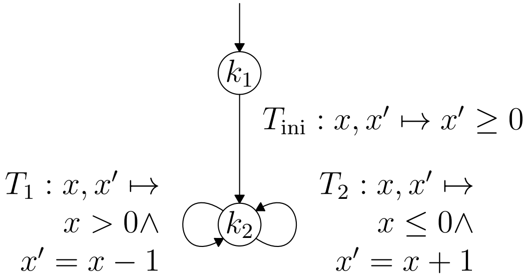
A global state of α is a couple $q = (k, v)$ where

- $k \in K$ is a control point of α.
- $v \in \mathrm{Val}$ is a valuation of Var.

q is initial if $k = k_{\mathrm{ini}}$.

$q = (k, v) \rightarrow q' = (k', v')$ iff there is a transition (k, T, k') such as $T(v, v')$.

Example



State $(k_2, 2)$ reachable through trace

$$(k_1, -6) \to (k_2, 4) \to (k_2, 3) \to (k_2, 2).$$

State $(k_2, -1)$ not reachable.

Control Node Splitting

Let $\text{Part} = P_1 \uplus \cdots \uplus P_m$ a partition of the domain of valuations Val s.t. every P_i is convex.

To split a control k in α across Part:

- Replace k with new controls $k_1, \ldots, k_n$.
- Delete each transition (k, T, k'). Add transitions (k_i, T_i, k') where $T_i(v, v') = T(v, v') \wedge v \in P_i$.
- Delete each transition (k', T, k). Add transitions (k', T_j, k_j) where $T_j(v, v') = T(v, v') \wedge v' \in P_j$.
- Delete each loop (k, T, k). Add transitions $(k_i, T_{i,j}, k_j)$ where $T_{i,j}(v, v') = T(v, v') \wedge v \in P_i \wedge v' \in P_j$.

Do not create unnecessary transitions & controls.

Control Node Splitting

Let $\text{Part} = P_1 \uplus \cdots \uplus P_m$ a partition of the domain of valuations Val s.t. every P_i is convex.

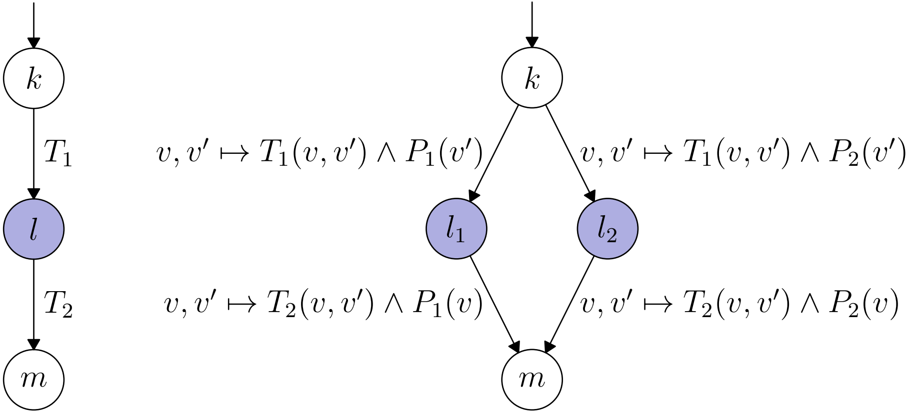
To split a control k in α across Part:

- Replace k with new controls $k_1, \ldots, k_n$.
- Delete each transition (k, T, k'). Add transitions (k_i, T_i, k') where $T_i(v, v') = T(v, v') \wedge v \in P_i$.
- Delete each transition (k', T, k). Add transitions (k', T_j, k_j) where $T_j(v, v') = T(v, v') \wedge v' \in P_j$.
- Delete each loop (k, T, k). Add transitions $(k_i, T_{i,j}, k_j)$ where $T_{i,j}(v, v') = T(v, v') \wedge v \in P_i \wedge v' \in P_j$.

Do not create unnecessary transitions & controls.

Equivalence theorems allows to use the resulting automaton to study the same properties.

Control Node Splitting



Parameters

The algorithm tends to create many controls & transitions, parameters must be chosen carefully.

Choice of controls

Split controls where accuracy loss is important, i.e. those with parallel loops.

Parameters

The algorithm tends to create many controls & transitions, parameters must be chosen carefully.

Choice of controls

Split controls where accuracy loss is important, i.e. those with parallel loops.

Choice of partition

Limit the size of the resulting automaton:

- Few partition components.
- Chosen s.t. some controls and some transitions are not created (preferentially those involved in loops).

Make the resulting transformers more precise.

Experiments

External tool, whose output is passed to the analyzer.

Partition choice: on a given control, determined by the truth values of all guards of transitions passing by the control.

Transformer T_1.

$g_1 = \{v \in \mathrm{Val} \mid \exists v' \in \mathrm{Val},\, T_1(v, v')\} = T_1$ projected on $x_1 \dots x_n$.

$$\underbrace{g_1 \wedge g_2 \wedge \dots}_{P_1} \quad \underbrace{\overline{g_1} \wedge g_2 \wedge \dots}_{P_2} \quad \underbrace{g_1 \wedge \overline{g_2} \wedge \dots}_{P_3} \quad \underbrace{\overline{g_1} \wedge \overline{g_2} \wedge \dots}_{P_4} \quad \dots$$

Experiments run on 71 previously published small scale transition systems ($\sim$ 1-10 controls, $\sim$ 2-10 transitions per control).

Considered successful if the expected invariant is found.

Experiments

With PIPS (revision 19448):

- 28 worked directly.
- $28 + 41 = 69$ worked with restructuration.
- 2 did not work.

Impact of restructuration: analysis 30 % slower, code 50 % bigger.

Experiments

With PIPS (revision 19448):

- 28 worked directly.
- $28 + 41 = 69$ worked with restructuration.
- 2 did not work.

Impact of restructuration: analysis 30 % slower, code 50 % bigger.

With ASPIC 3.1 (classical LRA with widening + accelerations):

- 44 worked directly.
- $44 + 21 = 65$ worked with restructuration
- 6 did not work.

1 fails in both.

Future Work

Performance issues:

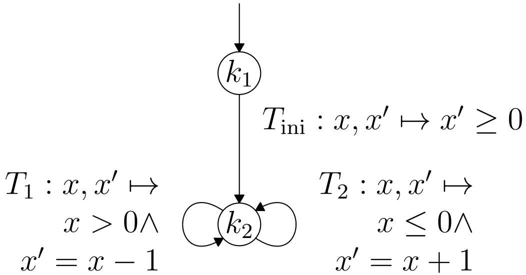
- The restructuration tends to create many controls and transitions, which limits its scope to small-scale systems.

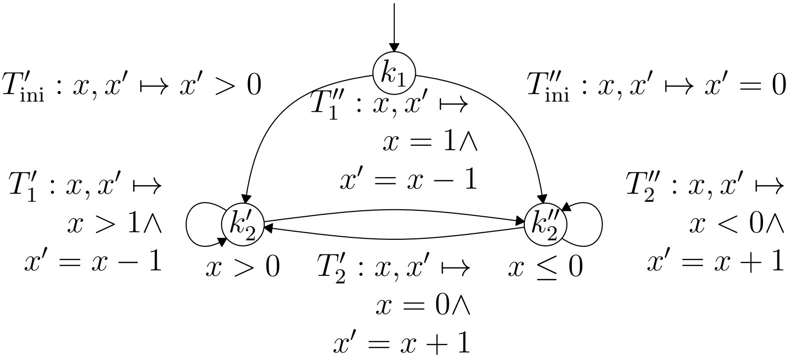
Suitability issues:

- Usually, better results with a manually chosen partition.
- Restructuration makes things worse on vicious systems.

$\Rightarrow$ Find better partition strategies, handle a wider range of systems.

Thank you.


k_1
$T_{\text{ini}} : x, x' \mapsto x' \geq 0$
$T_1 : x, x' \mapsto$
$x > 0 \wedge$
$x' = x - 1$
k_2
$T_2 : x, x' \mapsto$
$x \leq 0 \wedge$
$x' = x + 1$


k_1
$T'_{\text{ini}} : x, x' \mapsto x' > 0$
$T''_{\text{ini}} : x, x' \mapsto x' = 0$
$T''_1 : x, x' \mapsto$
$x = 1 \wedge$
$x' = x - 1$
$T'_1 : x, x' \mapsto$
$x > 1 \wedge$
$x' = x - 1$
k'_2
k''_2
$T''_2 : x, x' \mapsto$
$x < 0 \wedge$
$x' = x + 1$
$x > 0$
$T'_2 : x, x' \mapsto$
$x = 0 \wedge$
$x' = x + 1$
$x \leq 0$

Contents