Static Analysis of Control-Command Systems: Floating-Point and Integer Invariants

Vivien Maisonneuve

Thesis supervisors:
Olivier Hermant  François Irigoin

MINES ParisTech

Thesis defense
Paris, February 6, 2015
Control-Command Systems

A control-command system is a system that regulates the behavior of a device.
Control-Command Systems

A control-command system is a system that regulates the behavior of a device.

![Diagram of control-command system with device, control system, sensors, and actuators.]
Control-Command Systems

Wide spectrum of control-command systems

Critical applications $\Rightarrow$ need for reliability
Development Constraints

Often implemented on embedded systems

Extra constraints:

- reactivity (real time)
- autonomy
- cost
- energy, memory
Description Levels

Formalization

1. system conception
2. constraint specification
3. model of the environment (differential equations)
4. control theory: mathematical proofs of system behavior (stability)

MATLAB, Simulink
# Description Levels

<table>
<thead>
<tr>
<th>Formalization</th>
<th>Realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 system conception</td>
<td>low-level C program</td>
</tr>
<tr>
<td>2 constraint specification</td>
<td>- thousands of LOC</td>
</tr>
<tr>
<td>3 model of the environment (differential equations)</td>
<td>- computations decomposed into elementary operations</td>
</tr>
<tr>
<td>4 control theory: mathematical proofs of system behavior (stability)</td>
<td>- management of sensors and actuators</td>
</tr>
<tr>
<td>MATLAB, Simulink</td>
<td>- floating-point numbers</td>
</tr>
<tr>
<td>GCC, CLANG</td>
<td></td>
</tr>
</tbody>
</table>
Description Levels

**Formalization**

1. system conception
2. constraint specification
3. model of the environment (differential equations)
4. control theory: mathematical proofs of system behavior (stability)

MATLAB, Simulink

**Realization**

low-level C program

- thousands of LOC
- computations decomposed into elementary operations
- management of sensors and actuators
- floating-point numbers

GCC, CLANG

Step-by-step refinements

How to ensure that the executed program is correct?
Programming Critical Systems

Combination of different methods:

- documentation and specification of all software components
- coding standard to limit “dangerous” programming techniques (dynamic allocation, global variables…)
- extensive testing

Successes in implementing critical software:

- zero bug in space shuttle code (400 KLOC)
- after decades of civil aviation, no human casualty due to a defective software
Drawbacks

Long and expensive development process:

- 70 % of software development cost for Boeing 777, 25 % of the total development cost
- development constraint sometimes poorly respected: electronic throttle control defect in Toyota Camry vehicles

Exhaustive testing is impossible: can miss bugs.

⇒ Interest in formal methods to provide mathematical insurances of correctness.
Output after formalization: MATLAB pseudocode of a discrete-time controller with a frequency

repeat every 10ms:
  input command
  input sensors
  compute response
  send response to actuators
Output after formalization: MATLAB pseudocode of a discrete-time controller with a frequency and a stability proof

repeat every 10ms:
  input command // property $P_1$
  input sensors // property $P_2$
  compute response // property $P_3$
  send response to actuators // property $P_4$

"During execution, controller state variables $\in$ stability domain": ... ☐

Numerical invariants: mathematical properties on program variables, at each location in the code
Problems & Contributions

Formalization

C code implementation

1 Stability proof on real numbers
   Implementation on machine arithmetic
   • numeric constants
   • rounding errors in computations
Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework

Transpose code + invariants in two steps

Real (MATLAB)

\[
\% \quad d \\
\% \quad i \\
\% \quad d' = \theta (d, i)
\]
Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework

Transpose code + invariants in two steps

Real (MATLAB)

\[
\% d \\
\% i \\
\% d' = \theta(d, i)
\]

Intermediate

\[
\% \tilde{d} \\
\% \tilde{i} \\
\% \tilde{d}' = \theta(\tilde{d}, \tilde{i})
\]

Constants
Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework
Transpose code + invariants in two steps

Real (MATLAB)
\[
\% \quad d \\
\% \quad i \\
\% \quad d' = \theta(d, i)
\]

Intermediate
\[
\% \quad \tilde{d} \\
\% \quad \tilde{i} \\
\% \quad \tilde{d}' = \theta(\tilde{d}, \tilde{i})
\]

Machine (C)
\[
\% \quad \bar{d} \\
\% \quad \bar{i} \\
\% \quad \bar{d}' \supset \theta(\bar{d}, \bar{i}) \oplus \varepsilon
\]

Constants
Functions
Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework

Transpose code + invariants in two steps

Real (MATLAB)

\[
\% \quad d \\
\quad i \\
\% \quad d' = \theta(d, i)
\]

Intermediate

\[
\% \quad \tilde{d} \\
\quad \tilde{i} \\
\% \quad \tilde{d}' = \theta(\tilde{d}, \tilde{i})
\]

Machine (C)

\[
\% \quad \bar{d} \\
\quad \bar{i} \\
\% \quad \bar{d}' \supset \theta(\bar{d}, \bar{i}) \oplus \varepsilon
\]

Implementation

- Linear systems
- Floating-point & fixed-point arithmetic, parametric precision
- [Feron ICSM’10]: open-loop 32 bits, closed-loop 117 bits
Problems & Contributions

Formalization → C code implementation

2. Data acquisition through interrupt handlers outside the main loop

```c
SIGNAL (SIG_COMMAND) { ... }
SIGNAL (SIG_SENSOR1) { ... }
void main_loop() { ... }
```

⇒ different code structure
⇒ computation of global invariants to check the program runs at the correct frequency

New invariant computation techniques
- transformers
- benchmark to compare tools and algorithms
Outline

1 Transformers: Scalability and Accuracy

2 Improvements in Transformer Computation
   - Control-Path Transformers
   - Iterative Analysis
   - Arbitrary-Precision Numbers

3 Model Restructuring
   - State Splitting
   - State Merge
Linear Relation Analysis (LRA)
[Cousot & Halbwachs POPL’78]

// invariant (precondition)
program instruction
// invariant (postcondition)

invariant = system of affine (in)equalities = convex polyhedron

\[
\begin{align*}
\text{x}_1 + 7 & \geq 2\text{x}_2 \\
\text{x}_2 & \geq 5 - \text{x}_1 \\
\text{x}_2 & \geq 1 \\
\end{align*}
\]

Good trade-off: computational cost vs. accuracy
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    int n = 0;
    while (rand())
        if (rand())
            if (n < 60) n++;
        else n = 0;
}
```

• Propagation
• Branch output $P_6$: either $P_4$ or $P_5$
• Branch output $P_7$: $P_7 = P_2 \sqcup P_6$: $0 \leq n \leq 1$
• Loop invariant: $P_2$ entering the loop $P_7$ after one iteration
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // $P_0 : true$
    int n = 0;

    while (rand())
    {
        if (rand())
        {
            if (n < 60) n++;
        }
        else n = 0;
    }
}
```

- Propagation

• Propagation

$P_7 = P_2 \sqcup P_6 : 0 \leq n \leq 1$

• Loop invariant: $P_2$ entering the loop
$P_7$ after one iteration
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // P₀ : true
    int n = 0;
    // P₁ : n = 0
    while (rand())
        if (rand())
            if (n < 60) n++;
        else n = 0;
}
```

- Propagation

• Propagation

  P₆: either P₄ or P₅

  P₇: P₂ ⊔ P₆: 0 ≤ n ≤ 1

• Loop invariant: P₂ entering the loop

  P₇ after one iteration
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // $P_0$: true
    int n = 0;
    // $P_1$: $n = 0$
    while (rand())
        // $P_2$: $n = 0$
        if (rand())
            if (n < 60) n++;
        else n = 0;
}
```

- Propagation

$P_6$: either $P_4$ or $P_5$

$P_7$: $P_2 \sqcup P_6$: $0 \leq n \leq 1$

- Loop invariant: $P_2$: entering the loop $P_7$: after one iteration
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

void foo() {
    // $P_0$ : true
    int n = 0;
    // $P_1$ : $n = 0$
    while (rand())
        // $P_2$ : $n = 0$
        if (rand())
            // $P_3$ : $n = 0$
            if (n < 60) n++;
        else n = 0;
}

• Propagation

• Branch output

$P_6$:
either $P_4$ or $P_5$

• Branch output

$P_7$:
$P_7 = P_2 \sqcup P_6$:
0 $\leq n \leq 1$

• Loop invariant:
$P_2$ entering the loop
$P_7$ after one iteration
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // P0 : true
    int n = 0;
    // P1 : n = 0
    while (rand())
        // P2 : n = 0
        if (rand())
            // P3 : n = 0
            if (n < 60) n++;
            // P4 : n = 1
        else n = 0;
}
```

- **Propagation** in each branch
void foo() {
    // $P_0$: true
    int n = 0;
    // $P_1$: $n = 0$
    while (rand())
        // $P_2$: $n = 0$
        if (rand())
            // $P_3$: $n = 0$
            if (n < 60) n++;
            // $P_4$: $n = 1$
        else n = 0;
        // $P_5$: $\emptyset$
}

- **Propagation** in each branch
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // P0: true
    int n = 0;
    // P1: n = 0
    while (rand())
        // P2: n = 0
        if (rand())
            // P3: n = 0
            if (n < 60) n++;
            // P4: n = 1
        else n = 0;
        // P5: ∅
    // P6: ?
}
```

- Propagation in each branch
- Branch output $P_6$: either $P_4$ or $P_5$
void foo() {
    // $P_0$: true
    int n = 0;
    // $P_1$: $n = 0$
    while (rand())
        // $P_2$: $n = 0$
        if (rand())
            // $P_3$: $n = 0$
            if (n < 60) n++;
            // $P_4$: $n = 1$
        else n = 0;
        // $P_5$: $\emptyset$
    // $P_6$: $n = 1$
}

- Propagation in each branch
- Branch output $P_6$:
  either $P_4$ or $P_5$
  $P_6 = P_4 \sqcup P_5 : n = 1$
void foo() {
    // P₀: true
    int n = 0;
    // P₁: n = 0
    while (rand())
        // P₂: n = 0
        if (rand())
            // P₃: n = 0
            if (n < 60) n++;
            // P₄: n = 1
        else n = 0;
        // P₅: ∅
    // P₆: n = 1
    // P₇: 0 ≤ n ≤ 1
}

- Propagation in each branch
- Branch output $P₆$:
  either $P₄$ or $P₅$
  $P₆ = P₄ \sqcup P₅ : n = 1$
- Branch output $P₇$:
  $P₇ = P₂ \sqcup P₆ : 0 \leq n \leq 1$
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // P₀ : true
    int n = 0;
    // P₁ : n = 0
    while (rand())
        // P₂ : n = 0
        if (rand())
            // P₃ : n = 0
            if (n < 60) n++;
            // P₄ : n = 1
        else n = 0;
        // P₅ : ∅
        // P₆ : n = 1
        // P₇ : 0 ≤ n ≤ 1
}
```

- Propagation in each branch
- Branch output $P₆$:
  - either $P₄$ or $P₅$
  - $P₆ = P₄ ⊔ P₅ : n = 1$
- Branch output $P₇$:
  - $P₇ = P₂ ⊔ P₆ : 0 ≤ n ≤ 1$
- Loop invariant:
  - $P₂$ entering the loop
  - $P₇$ after one iteration
Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS’12]

```c
void foo() {
    // P0 : true
    int n = 0;
    // P1 : n = 0
    while (rand())
        // P2 : n = 0
        if (rand())
            // P3 : n = 0
            if (n < 60) n++;
            // P4 : n = 1
        else n = 0;
        // P5 : ∅
    // P6 : n = 1
    // P7 : 0 ≤ n ≤ 1
}
```

- Propagation in each branch
- Branch output $P_6$: either $P_4$ or $P_5$
  
  $P_6 = P_4 \sqcup P_5 : n = 1$

- Branch output $P_7$:
  
  $P_7 = P_2 \sqcup P_6 : 0 \leq n \leq 1$

- Loop invariant:
  
  $P_2$ entering the loop $P_7$ after one iteration

Widening:

$P^* = P_2 \triangledown P_7 : 0 \leq n$

did not found $n \leq 60$
Transformers

Alternate approach:

1. Abstraction of each program instruction, block, function by a transformer = polyhedral approximation of the transfer function

2. Invariant propagation using transformers

Used in PIPS
Transformers

Alternate approach:

1. Abstraction of each program instruction, block, function by a transformer = polyhedral approximation of the transfer function
2. Invariant propagation using transformers

Used in PIPS

Pros:

- Interprocedural analysis
- Nested loops

⇒ Supports large applications

Cons:

- Double abstraction ⇒ less accurate
- $2 \times$ variables ⇒ complexity cost
void foo() {

    int n = 0;

    while (rand())
        if (rand())
            if (n < 60) n++;
        else n = 0;
}

• Elementary instructions
• Compound statements
• Transitive closure
[Ancourt et al. NSAD'10]

• Invariant propagation using transformers
Transformers

```c
void foo() {

    int n = 0;  // $T_0 : n' = 0$  ($n'$ : new value of $n$)

    while (rand())

        if (rand())

            if (n < 60) n++;

        else n = 0;

}
```

- Elementary instructions
void foo() {

    int n = 0; // $T_0: n' = 0$ ($n'$: new value of $n$)

    while (rand())

        if (rand())

            if (n < 60) n++; // $T_4: n < 60, n' = n + 1$

        else n = 0;
}

- Elementary instructions
void foo() {

    int n = 0; // $T_0: n' = 0$ ($n'$: new value of $n$)

    while (rand())

        if (rand())

            if (n < 60) n++; // $T_4: n < 60, n' = n + 1$
            else n = 0; // $T_5: n \geq 60, n' = 0$

}
Transformers

```c
void foo() {

    int n = 0; // $T_0 : n' = 0$ ($n'$: new value of $n$)

    while (rand())

        if (rand())
            // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
            if (n < 60) n++; // $T_4 : n < 60, n' = n + 1$
        else n = 0; // $T_5 : n \geq 60, n' = 0$

}

- Elementary instructions
- Compound statements
```
Transformers

```c
void foo() {

    int n = 0; // T_0: n' = 0 (n': new value of n)

    while (rand())
        if (rand()) // T_2 = T_3 [\square] \text{Id}: n' \leq n + 1
            // T_3 = T_4 \sqcup T_5: n' \leq 60, n' \leq n + 1
            if (n < 60) n++; // T_4: n < 60, n' = n + 1
        else n = 0; // T_5: n \geq 60, n' = 0

}
```

- Elementary instructions
- Compound statements
Transformers

void foo() {

    int n = 0;  // $T_0: n' = 0$ (n' : new value of n)

    while (rand())  // $T_1 = T_2^*: true$

        if (rand())  // $T_2 = T_3 \sqcup \text{Id}: n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5: n' \leq 60, n' \leq n + 1$
            if (n < 60) n++;  // $T_4: n < 60, n' = n + 1$
        else n = 0;  // $T_5: n \geq 60, n' = 0$

}

- Elementary instructions
- Compound statements
- Transitive closure
  [Ancourt et al. NSAD’10]
void foo() {
    // $P_0 : \text{true}$
    int n = 0; // $T_0 : n' = 0$ (n' : new value of n)

    while (rand()) // $T_1 = T_2^* : \text{true}$

        if (rand()) // $T_2 = T_3 \uplus \text{Id} : n' \leq n + 1$
            // $T_3 = T_4 \uplus T_5 : n' \leq 60, n' \leq n + 1$
            if (n < 60) n++; // $T_4 : n < 60, n' = n + 1$
        else n = 0; // $T_5 : n \geq 60, n' = 0$
}

- Elementary instructions
- Compound statements
- Transitive closure
  [Ancourt et al. NSAD'10]
- Invariant propagation using transformers
Transformers & Invariants

```c
void foo() {
    // P₀: true
    int n = 0; // T₀: n' = 0 (n': new value of n)
    // P₁: n = 0
    while (rand()) // T₁ = T₂*: true

        if (rand()) // T₂ = T₃ ▽ Id: n' ≤ n + 1
            // T₃ = T₄ ▽ T₅: n' ≤ 60, n' ≤ n + 1
        if (n < 60) n++; // T₄: n < 60, n' = n + 1
        else n = 0; // T₅: n ≥ 60, n' = 0

}
```

- Elementary instructions
- Compound statements
- Transitive closure
  [Ancourt et al. NSAD’10]
- Invariant propagation using transformers
void foo() {
    // $P_0$: true
    int n = 0; // $T_0: n' = 0$ (n': new value of n)
    // $P_1: n = 0$
    while (rand()) // $T_1 = T_2^*: true$
        // $P_6: true$
        if (rand()) // $T_2 = T_3 \sqcup \text{Id}: n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5: n' \leq 60, n' \leq n + 1$
            if (n < 60) n++; // $T_4: n < 60, n' = n + 1$
        else n = 0; // $T_5: n \geq 60, n' = 0$
}

- Elementary instructions
- Compound statements
- Transitive closure
  [Ancourt et al. NSAD’10]

- Invariant propagation using transformers
Evaluation of the Transformer Approach

Questions

1. Ability to deal with complex programs (computation cost)
2. Accuracy of generated invariants
Evaluation of the Transformer Approach

Questions

1. Ability to deal with complex programs (computation cost)
2. Accuracy of generated invariants

ALICe: a framework for Affine Loop Invariant Computation

- Compare several techniques & tools to compute affine loop invariants on a common set of previously published examples
- 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
Tool Selection

Comparison of

- **PIPS:**
  Transformer-based

with available, state-of-the-art tools

- **ASPIC:**
  Classic LRA + accelerations

- **ISL:**
  Presburger-equivalent library with powerful transitive closure heuristics

- **PAGAI:**
  Classic LRA + decision procedures (SMT-solving)
Tool Selection

Comparison of

- **PIPS:**
  Transformer-based
  C code

with available, state-of-the-art tools

- **ASPIC:**
  Classic LRA + accelerations
  Finite state machine in FSM format

- **ISL:**
  Presburger-equivalent library with powerful transitive closure heuristics
  Transitive closure of transition relation

- **PAGAI:**
  Classic LRA + decision procedures (SMT-solving)
  LLVM IR compiled from C code
Input Format
Test cases written in FSM (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};
}

Simple, existing basis 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 that the invariant does not reach the error region

$\Rightarrow$ Several wrappers and format conversion tools involved

Challenge: generate structured C code from a state machine
Impact of Cycle Nesting on Convergence Time

Analysis of loop nests:

```c
for (i1 = 0; i1 < b1; i1++)
    for (i2 = 0; i2 < b2; i2++)
    ...
```

Time measurements: Intel i7-2600, 3.40 GHz, 16 GB RAM

- `time` command for ASPIC, ISL (internal formats), PAGAI (CLANG)
- `LOG_TIMINGS` for PIPS (transformers and preconditions, no parsing)

<table>
<thead>
<tr>
<th>Depth</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPIC</td>
<td>0.037</td>
<td>0.043</td>
<td>0.040</td>
<td>0.053</td>
<td>0.047</td>
<td>0.063</td>
<td>0.067</td>
<td>0.087</td>
<td>0.100</td>
</tr>
<tr>
<td>ISL</td>
<td>0.000</td>
<td>0.010</td>
<td>0.037</td>
<td>0.083</td>
<td>0.370</td>
<td>0.853</td>
<td>1.197</td>
<td>7.927</td>
<td>5.713</td>
</tr>
<tr>
<td>PAGAI</td>
<td>0.067</td>
<td>0.187</td>
<td>0.420</td>
<td>0.797</td>
<td>1.373</td>
<td>2.260</td>
<td>3.620</td>
<td>5.780</td>
<td>9.643</td>
</tr>
<tr>
<td>PIPS</td>
<td>0.004</td>
<td>0.009</td>
<td>0.015</td>
<td>0.021</td>
<td>0.030</td>
<td>0.039</td>
<td>0.053</td>
<td>0.071</td>
<td>0.090</td>
</tr>
</tbody>
</table>
Interprocedural Analysis vs. Inlining

```c
void mm(int l, int n, int m,
        float A[l][m], float B[l][n], float C[n][m]) {
    // naive matrix multiplication
    // A = B * C
    ...
}

void mp(int n, int p,
        float A[n][n], float B[n][n]) {
    // matrix exponentiation
    // A = B\(^p\)
    ...
    mn(...);
    ...
}
```
### Interprocedural Analysis vs. Inlining

```c
int main(void) {
    ...
    mp(...);
    mp(...);
    ...
}
```

<table>
<thead>
<tr>
<th></th>
<th>Inlining</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPIC</td>
<td>Yes</td>
<td>0.043</td>
<td>0.061</td>
<td>0.087</td>
<td>0.108</td>
<td>0.149</td>
</tr>
<tr>
<td>ISL</td>
<td>Yes</td>
<td>261.810</td>
<td>274.580</td>
<td>370.960</td>
<td>413.300</td>
<td>456.360</td>
</tr>
<tr>
<td>PAGAI</td>
<td>Yes</td>
<td>1.417</td>
<td>5.680</td>
<td>14.677</td>
<td>30.007</td>
<td>53.247</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>0.980</td>
<td>1.383</td>
<td>2.030</td>
<td>2.990</td>
<td>4.467</td>
</tr>
<tr>
<td>PIPS</td>
<td>Yes</td>
<td>0.043</td>
<td>0.063</td>
<td>0.084</td>
<td>0.108</td>
<td>0.127</td>
</tr>
<tr>
<td></td>
<td>No</td>
<td>0.048</td>
<td>0.049</td>
<td>0.048</td>
<td>0.050</td>
<td>0.051</td>
</tr>
</tbody>
</table>
Accuracy Results with ALICe

What about accuracy?

Out of 102 test cases

- ASPIC: 75 test cases correctly analyzed
- ISL: 63
- PIPS: 43

[Maisonneuve et al., WING’14]
Accuracy Results with ALICE

No tool is strictly better

No trend for invariant accuracy

- ISL good with concurrent loops, unlike PIPS
- ISL slow on large control structures
- ASPIC in difficulty with complex formulæ (no acceleration)
Evaluation & Shortcomings

Evaluation of PIPS approach

- Effective for large programs with function calls, nested loops
- Lacks accuracy for transition systems challenging invariant generation

Sources of inaccuracy

- Multiple control paths nested within loops:
  convex hulls + transitive closures
- Arithmetic overflows

⇒ Improvements needed in transformer computation
(Implemented in PIPS)
Control-Path Transformers

[Maissenneuve et al. NSAD'14]

while (true)
{

    if ... // $T_1$
    else ... // $T_2$

}
Control-Path Transformers

[Maisonneuve et al. NSAD’14]

while (true)
{
   // $T = T_1 \sqcup T_2$

   if ... // $T_1$
   else ... // $T_2$
}

Use alternate formula:

$$P'' = P \sqcup T_1 + (P) \sqcup T_2 + (P) \sqcup (T_1 \circ T_2) (P) \sqcup (T_2 \circ T_1) (P) \sqcup (T_1 + \circ T_2) (P) \sqcup (T_2 + \circ T_1) (P)$$

Convex hulls are postponed, performed on invariants instead of transformers $\Rightarrow$ more information is preserved:

$P'' \subset P'$. 

27 / 41
Control-Path Transformers
[Maisonneuve et al. NSAD’14]

\[ \text{while (true)} \quad // \quad T^* = (T_1 \sqcup T_2)^* \]
\[ \{ \quad // \quad T = T_1 \sqcup T_2 \]

\[ \quad \text{if ...} \quad // \quad T_1 \]
\[ \quad \text{else ...} \quad // \quad T_2 \]
\}
Control-Path Transformers

[Maisonneuve et al. NSAD’14]

// P
while (true) // $T^* = (T_1 \sqcup T_2)^*$
{
  // $T = T_1 \sqcup T_2$
  // $P' = T^*(P)$
  if ... // $T_1$
  else ... // $T_2$
}

Use alternate formula:

$P'' = P \sqcup T_1 + (P) \sqcup T_2 + (P) \sqcup (T_1 \circ T_2)(P) \sqcup (T_2 \circ T_1)(P) \sqcup (T_1 \circ T_2^*)(P) \sqcup (T_2 \circ T_1^*)$
Control-Path Transformers
[Maisonneuve et al. NSAD’14]

// P
while (true) // \( T^\ast = (T_1 \sqcup T_2)^\ast \)
{ // \( T = T_1 \sqcup T_2 \)
  // \( P' = T^\ast(P) \)
  if ... // \( T_1 \)
  else ... // \( T_2 \)
}

Use alternate formula:

\[
P'' = P \sqcup T_1^+(P) \sqcup T_2^+(P) \sqcup (T_1 \circ T_2)(P) \sqcup (T_2 \circ T_1)(P) \sqcup (T_1^+ \circ T_2 \circ T^\ast)(P) \sqcup (T_2^+ \circ T_1 \circ T^\ast)(P)
\]

Convex hulls are postponed, performed on invariants instead of transformers
\( \Rightarrow \) more information is preserved: \( P'' \subset P' \)
Control-Path Transformers

```c
void foo() {
    // P0 : true
    int n = 0; // T0 : n' = 0 (n' : new value of n)
    // P1 : n = 0
    while (true) // T1 = T2* : true
        // P6 = T2*(P1) : true
        if (rand()) // T2 = T3 ▽ Id : n' ≤ n + 1
            // T3 = T4 ▽ T5 : n' ≤ 60, n' ≤ n + 1
            if (n < 60) n++; // T4 : n' ≤ 60, n' = n + 1
        else n = 0; // T5 : n > 60, n' = 0
}
```
Control-Path Transformers

```c
void foo() {
    // P₀: true
    int n = 0; // T₀: n' = 0 (n': new value of n)
    // P₁: n = 0
    while (true) // T₁ = T₂*: true
        // P₆ = T₂*(P₁): true
        if (rand()) // T₂ = T₃ □ Id: n' ≤ n + 1
            // T₃ = T₄ □ T₅: n' ≤ 60, n' ≤ n + 1
            if (n < 60) n++; // T₄: n' ≤ 60, n' = n + 1
        else n = 0; // T₅: n > 60, n' = 0
}
```

With convex hulls in the precondition space:

\[
P'_6 = P_1 □ T_3^+(P_1) □ Id^+(P_1) □ (T_3 ◦ Id)(P_1) □ (Id ◦ T_3)(P_1) □ (T_3^+ ◦ Id ◦ (T_3 □ Id))(P_1) □ (Id^+ ◦ T_3 ◦ (T_3 □ Id))(P_1)
\]
Control-Path Transformers

```c
void foo() {
    // P0: true
    int n = 0; // T0: n' = 0  (n': new value of n)
    // P1: n = 0
    while (true) // T1 = T2*: true
        // P6 = T2*(P1): true
        if (rand()) // T2 = T3 ▽ Id: n' ≤ n + 1
            // T3 = T4 ▽ T5: n' ≤ 60, n' ≤ n + 1
            if (n < 60) n++; // T4: n' ≤ 60, n' = n + 1
        else n = 0; // T5: n > 60, n' = 0
}
```

With convex hulls in the precondition space:

\[
P'_6 = P_1 \sqcup T_3^+(P_1) \sqcup Id^+(P_1) \sqcup (T_3 \circ Id)(P_1) \sqcup (Id \circ T_3)(P_1) \sqcup (T_3^+ \circ Id \circ (T_3 \sqcup Id))(P_1) \sqcup (Id^+ \circ T_3 \circ (T_3 \sqcup Id))(P_1) = P_1 \sqcup T_3^+(P_1) \sqcup (T_3^+ \circ (T_3 \sqcup Id))(P_1)
\]
Control-Path Transformers

```c
void foo() {
    // P0 : true
    int n = 0; // T0 : n' = 0 (n' : new value of n)
    // P1 : n = 0
    while (true) // T1 = T2* : true
        // P6 = T2*(P1) : true
        if (rand()) // T2 = T3 ⊔ Id : n' ≤ n + 1
            // T3 = T4 ⊔ T5 : n' ≤ 60, n' ≤ n + 1
            if (n < 60) n++; // T4 : n' ≤ 60, n' = n + 1
        else n = 0; // T5 : n > 60, n' = 0
}
```

With convex hulls in the precondition space:

\[
P'_6 = P_1 ⊔ T_3^+(P_1) ⊔ \text{Id}^+(P_1) ⊔ (T_3 \circ \text{Id})(P_1) ⊔ (\text{Id} \circ T_3)(P_1) ⊔ \\
(T_3^+ \circ \text{Id} \circ (T_3 \square \text{Id}))(P_1) ⊔ (\text{Id}^+ \circ T_3 \circ (T_3 \square \text{Id}))(P_1)
\]

\[
= P_1 ⊔ T_3^+(P_1) ⊔ (T_3^+ \circ (T_3 \square \text{Id}))(P_1)
\]

\[P'_6 : 0 \leq n \leq 60\]
Iterative Analysis

- At iteration 1, compute transformers and invariants as usual

[Dillig et al., OOPSLA'13]

```c
void bar(float x) {
    int i, j = 1, a = 0, b = 0;
    i = 0;
    while (rand()) {
        a++; b += j-i; i += 2;
        if (i % 2 == 0) j += 2;
        else j++;
    }
}
```
Iterative Analysis

- At iteration 1, compute transformers and invariants as usual

[Dillig et al., OOPSLA’13]

```c
void bar(float x) {
    int i, j = 1, a = 0, b = 0;
    i = 0;
    while (rand()) {
        // P1: 2a = i, j ≤ 2a + 1, a + 1 ≤ j
        a++; b += j-i; i += 2;
        if (i % 2 == 0) j += 2;
        else j++;
    }
}
```
Iterative Analysis

- At iteration 1, compute transformers and invariants as usual
- At iteration \( n + 1 \), sharpen transformers with invariants found at iteration \( n \), then recompute invariants

[Dillig et al., OOPSLA’13]

```c
void bar(float x) {
    int i, j = 1, a = 0, b = 0;
    i = 0;
    while (rand()) {
        // \( P_1 : 2a = i, j \leq 2a + 1, a + 1 \leq j \)
        a++; b += j-i; i += 2;
        if (i % 2 == 0) j += 2;
        else j++;
    }
}
```
Iterative Analysis

- At iteration 1, compute transformers and invariants as usual
- At iteration \( n + 1 \), sharpen transformers with invariants found at iteration \( n \), then recompute invariants

[Dillig et al., OOPSLA’13]

```c
void bar(float x) {
    int i, j = 1, a = 0, b = 0;
    i = 0;
    while (rand()) {
        // \( P_1 : 2a = i, j \leq 2a + 1, a + 1 \leq j \)
        // \( P_2 : 2a = i, 2a = j - 1, 0 \leq a, b \leq a \)
        a++; b += j-i; i += 2;
        if (i % 2 == 0) j += 2;
        else j++;
    }
}
```
Iterative Analysis

- At iteration 1, compute transformers and invariants as usual
- At iteration $n + 1$, sharpen transformers with invariants found at iteration $n$, then recompute invariants

[Dillig et al., OOPSLA’13]

```c
void bar(float x) {
    int i, j = 1, a = 0, b = 0;
    i = 0;
    while (rand()) {
        // $P_1$: $2a = i, j \leq 2a + 1, a + 1 \leq j$
        // $P_2$: $2a = i, 2a = j - 1, 0 \leq a, b \leq a$
        // $P_3$: $a = b, 2a = i, 2a = j - 1, 0 \leq a$
        a++; b += j-i; i += 2;
        if (i % 2 == 0) j += 2;
        else j++;
    }
}
```
Arbitrary-Precision Numbers

- Polyhedra with huge coefficients in intermediate computations
- Arithmetic overflow $\Rightarrow$ constraint dropped
- Less accurate invariant

GMP support added to Linear/C3, used by PIPS
### Experimental Results

Out of 102 test cases in ALICe:

<table>
<thead>
<tr>
<th>Options</th>
<th>None</th>
<th>CP</th>
<th>IA</th>
<th>CP-IA</th>
<th>CP-IA-MP</th>
<th>ASPIC</th>
<th>ISL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successes</td>
<td>43</td>
<td>69</td>
<td>45</td>
<td>72</td>
<td>73</td>
<td>75</td>
<td>63</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>6.1</td>
<td>7.8</td>
<td>18.5</td>
<td>19.6</td>
<td>151.4</td>
<td>10.9</td>
<td>35.5</td>
</tr>
</tbody>
</table>
Remaining Failures

- C code generated by ALICE from CFG may blow up exponentially
  Some cases work with native C encoding
- Cases require non-convex invariant
- No control-path transformers on inner loops or loops with many control paths
Sources of Approximations

For both classic LRA / transformers

- Loops (widening / transitive closure)
- Test branches (convex union)

Cumulative impact: multiple control paths nested within loops
Model Restructuring

Restructure the input model into an equivalent one

Formally, a model transformation is a function: $M_1 \xrightarrow{\text{transformation}} M_2$ 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
[Maisonneuve NSAD’11]

Designed to get rid of nodes with several self loops, difficult to analyze
Heuristic to split nodes according to loop guards
State Splitting
[Maisonneuve NSAD’11]

Designed to get rid of nodes with several self loops, difficult to analyze
Heuristic to split nodes according to loop guards

\[ k_1 \quad t_1 : x \geq 0? \quad k_2 \]
\[ t_2 : x \leq 0? \quad x++ \]
\[ t_3 : x \geq 1? \quad x++ \]

\[ k_1 \quad t_1 : x = 0? \quad k_2' \]
\[ t_1' : x \geq 1? \quad \]
\[ t_3 : x \geq 2? \quad x-- \]
\[ k_2'' \quad \]
\[ t_2 : x++ \]
\[ t_3 : x = 1? \quad x-- \]
\[ t_3' : x \geq 1? \quad \]
State Merge

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

**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
## 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>Direct</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successes</td>
<td>75</td>
<td>63</td>
<td>73</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>10.9</td>
<td>35.5</td>
<td>113.2</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>77</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>12.8</td>
<td>43.0</td>
<td>156.3</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>68</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>16.7</td>
<td>26.2</td>
<td>126.6</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>82</td>
</tr>
<tr>
<td>Time (s.)</td>
<td>11.3</td>
<td>40.8</td>
<td>146.3</td>
</tr>
</tbody>
</table>

### Analysis:
- 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
Comparative Results

- **Direct**
  - ASPIC: 24
  - ISL: 2
  - PIPS: 1

- **Split**
  - ASPIC: 19
  - ISL: 3
  - PIPS: 7

- **Merge**
  - ASPIC: 29
  - ISL: 1
  - PIPS: 11

- **Merge + Split**
  - ASPIC: 14
  - ISL: 3
  - PIPS: 14
Conclusion on LRA

Summary

- Transformer approach time-efficient for large pieces of code
- Lacks accuracy for challenging cases
- Improvements in loop invariant generation:
  - control-path transformers
  - iterative analysis
  - arbitrary-precision numbers
- Model restructuring, efficient for other tools too
- Comparable accuracy of PIPS with respect to ASPIC and ISL

Future Work

- Better support for C codes in ALICe
- More test cases
- More tools
- Improve invariant generation while avoiding exponential blowup
General Conclusion

Contributions

- Transposition of stability proof to floating-point arithmetic
- Output code analysis with LRA

To Be Done

- Further developments of contributions: scope, accuracy, performances
- Discretization: bounded computation time in the control loop
- Data acquisition: model interrupt handlers
Static Analysis of Control-Command Systems: Floating-Point and Integer Invariants

Vivien Maisonneuve

Thesis supervisors:
Olivier Hermant    François Irigoin

Thesis defense
Paris, February 6, 2015