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

Vivien Maisonneuve

Thesis supervisors:
Olivier Hermant François Irigoin

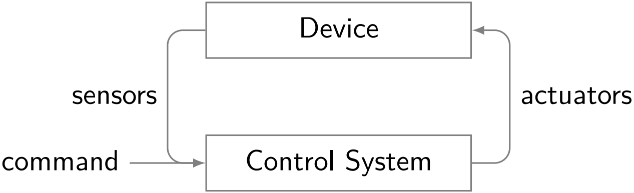
MINES ParisTech

Thesis defense

Paris, February 6, 2015

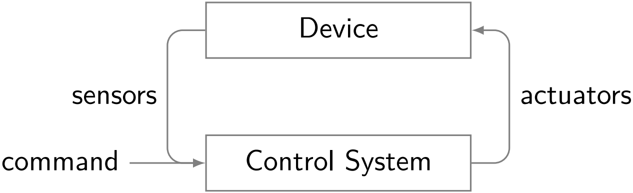
Control-Command Systems

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.



Device

sensors

actuators

command

Control System

Control-Command Systems

Wide spectrum of control-command systems



Critical applications ⇒ 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

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

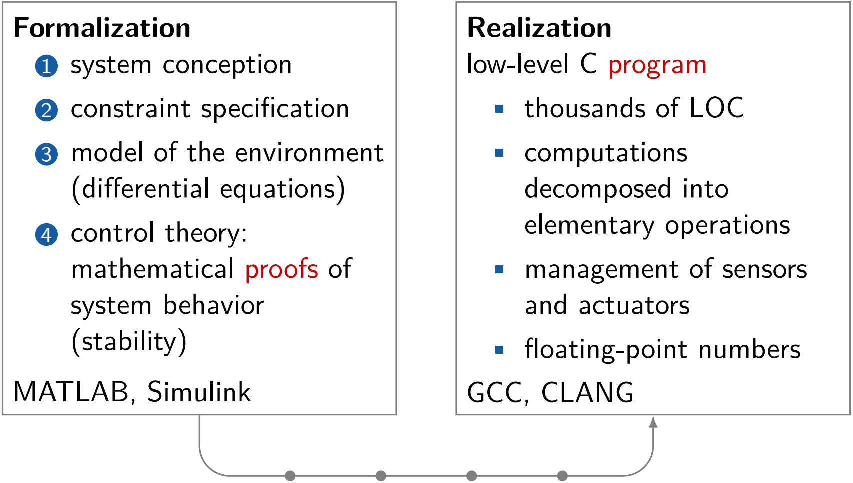
Realization

low-level C program

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

GCC, CLANG

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

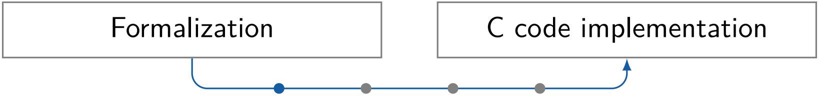
Long and expensive development process:

- 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.

Context



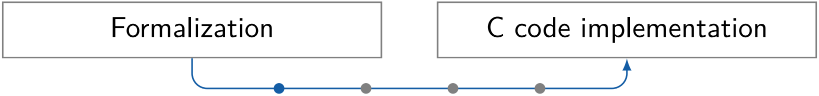
Formalization

C code implementation

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
```

Context



Formalization

C code implementation

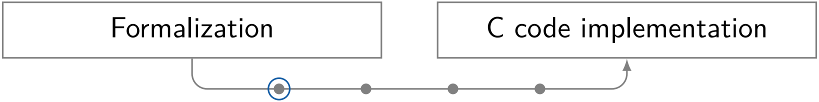
Output after formalization: MATLAB pseudocode of a discrete-time controller with a frequency and a stability proof

```
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": ... $\square$

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

Problems & Contributions



❶ 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

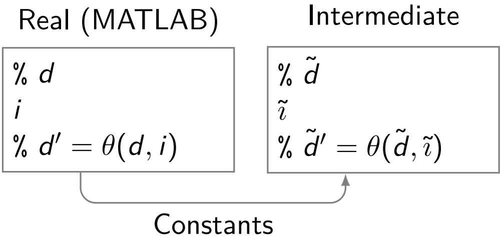
Real (MATLAB)

% d
i
% $d' = \theta(d, i)$

Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework

Transpose code + invariants in two steps



Real (MATLAB)

% d
i
% $d' = \theta(d, i)$

Intermediate

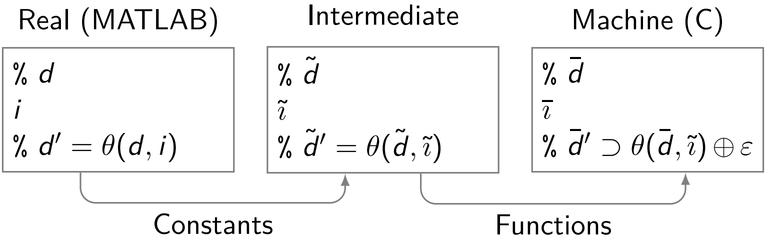
% $\tilde{d}$
$\tilde{\imath}$
% $\tilde{d}' = \theta(\tilde{d}, \tilde{\imath})$

Constants

Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework

Transpose code + invariants in two steps



Real (MATLAB)

% d
i
% $d' = \theta(d, i)$

Intermediate

% $\tilde{d}$
$\tilde{\imath}$
% $\tilde{d}' = \theta(\tilde{d}, \tilde{\imath})$

Machine (C)

% $\bar{d}$
$\bar{\imath}$
% $\bar{d}' \supset \theta(\bar{d}, \tilde{\imath}) \oplus \varepsilon$

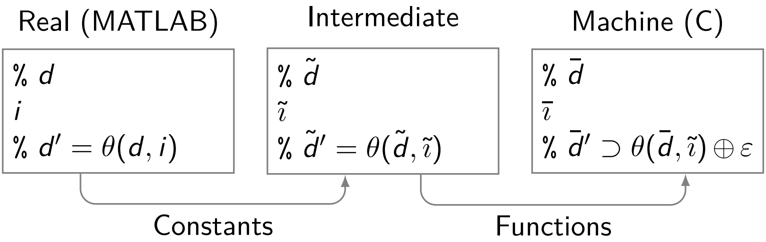
Constants

Functions

Transposition of Stability Proof to Machine Arithmetic

Theoretical Framework

Transpose code + invariants in two steps



Real (MATLAB)

% d
i
% $d' = \theta(d, i)$

Intermediate

% $\tilde{d}$
$\tilde{\imath}$
% $\tilde{d}' = \theta(\tilde{d}, \tilde{\imath})$

Machine (C)

% $\bar{d}$
$\bar{\imath}$
% $\bar{d}' \supset \theta(\bar{d}, \tilde{\imath}) \oplus \varepsilon$

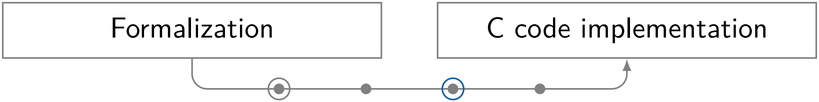
Constants

Functions

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

❷ Data acquisition through interrupt handlers outside the main loop

```
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

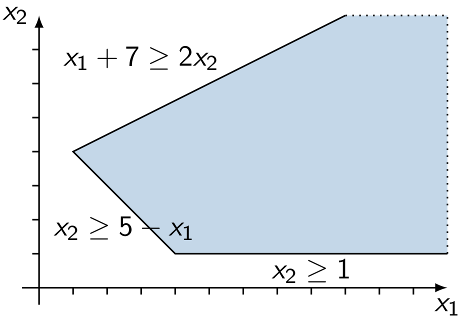
Outline

Linear Relation Analysis (LRA)

[Cousot & Halbwachs POPL'78]

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

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



Good trade-off: computational cost vs. accuracy

Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS'12]

```
void foo() {

    int n = 0;

    while (rand())

        if (rand())

            if (n < 60) n++;

            else n = 0;



}
```

Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS'12]

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

    while (rand())

        if (rand())

            if (n < 60) n++;

            else n = 0;



}
```

- Propagation

Linear Relation Analysis (LRA)

[Halbwachs & Henry SAS'12]

```
void foo() {
    // $P_0$ : true
    int n = 0;
    // $P_1$ : $n = 0$
    while (rand())

        if (rand())

            if (n < 60) n++;

            else n = 0;




}
```

- Propagation

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())

            if (n < 60) n++;

            else n = 0;




}
```

- Propagation

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

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++;
                // $P_4$ : $n = 1$
            else n = 0;




}
```

- Propagation in each branch

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++;
                // $P_4 : n = 1$
            else n = 0;
                // $P_5 : \emptyset$


}
```

- Propagation in each branch

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++;
                // $P_4 : n = 1$
            else n = 0;
                // $P_5 : \emptyset$
            // $P_6$ : ?

}
```

- Propagation in each branch
- Branch output P_6:
  either P_4 or P_5

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++;
                // $P_4 : n = 1$
            else n = 0;
                // $P_5 : \emptyset$
            // $P_6 : n = 1$

}
```

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

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++;
                // $P_4 : n = 1$
            else n = 0;
                // $P_5 : \emptyset$
            // $P_6 : n = 1$
        // $P_7 : 0 \leq n \leq 1$
}
```

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

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++;
                // $P_4 : n = 1$
            else n = 0;
                // $P_5 : \emptyset$
            // $P_6 : n = 1$
        // $P_7 : 0 \leq n \leq 1$
}
```

- 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

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++;
                // $P_4 : n = 1$
            else n = 0;
                // $P_5 : \emptyset$
            // $P_6 : n = 1$
        // $P_7 : 0 \leq n \leq 1$
}
```

- 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:

  $\boxed{P^* = P_2 \nabla P_7 : 0 \leq n}$

  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 × variables ⇒ complexity cost

Transformers

```
void foo() {

    int n = 0;

    while (rand())

        if (rand())

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

Transformers

```
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

Transformers

```
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

Transformers

```
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; // $T_5 : n \geq 60, n' = 0$
}
```

- Elementary instructions

Transformers

```
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

```
void foo() {

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

    while (rand())

        if (rand()) // $T_2 = T_3 \sqcup \mathrm{Id} : n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
            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; // $T_0 : n' = 0 \quad (n' : \text{new value of } n)$

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

        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]

Transformers & Invariants

```
void foo() {
    // $P_0$ : true
    int n = 0; // $T_0 : n' = 0$   ($n'$ : new value of $n$)

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

       if (rand()) // $T_2 = T_3 \sqcup \mathrm{Id} : n' \leq n + 1$
          // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
          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

Transformers & Invariants

```
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

        if (rand()) // $T_2 = T_3 \sqcup \mathrm{Id} : n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
            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

Transformers & Invariants

```
void foo() {
    // $P_0$ : true
    int n = 0; // $T_0 : n' = 0$   ($n'$ : new value of $n$)
    // $P_1 : n = 0$
    while (rand()) // $T_1 = T_2^* : \text{true}$
        // $P_6$ : 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

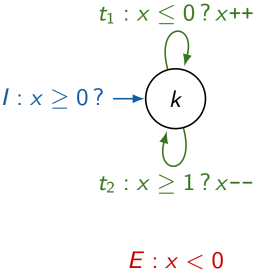
- PIPS:
  Transformer-based
  C code

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

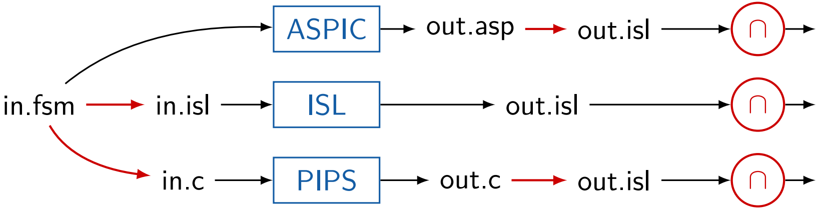
Test cases written in FSM (ASPIC format, introduced by FAST).



```
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

⇒ 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:

```
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)

Depth	1	2	3	4	5	6	7	8	9
ASPIC	0.037	0.043	0.040	0.053	0.047	0.063	0.067	0.087	0.100
ISL	0.000	0.010	0.037	0.083	0.370	0.853	1.197	7.927	5.713
PAGAI	0.067	0.187	0.420	0.797	1.373	2.260	3.620	5.780	9.643
PIPS	0.004	0.009	0.015	0.021	0.030	0.039	0.053	0.071	0.090

Interprocedural Analysis vs. Inlining

```
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

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

	Inlining	2	3	4	5	6
ASPIC	Yes	0.043	0.061	0.087	0.108	0.149
ISL	Yes	261.810	274.580	370.960	413.300	456.360
PAGAI	Yes	1.417	5.680	14.677	30.007	53.247
	No	0.980	1.383	2.030	2.990	4.467
PIPS	Yes	0.043	0.063	0.084	0.108	0.127
	No	0.048	0.049	0.048	0.050	0.051

Accuracy Results with ALICe

What about accuracy?

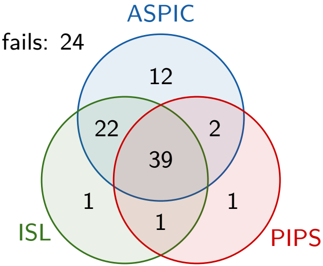
Out of 102 test cases

- 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

↯	ASPIC	ISL	PIPS
ASPIC	–	21	23
ISL	49	–	54
PIPS	33	23	–

- 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

$\Rightarrow$ Improvements needed in transformer computation

(Implemented in PIPS)

Control-Path Transformers

[Maisonneuve *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 ... // $T_2$
}
```

Control-Path Transformers

[Maisonneuve *et al.* NSAD'14]

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

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

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 ... // $T_2$
}
```

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:

$$P'' = P \sqcup {T_1}^+(P) \sqcup {T_2}^+(P) \sqcup (T_1 \circ T_2)(P) \sqcup (T_2 \circ T_1)(P) \sqcup \\ ({T_1}^+ \circ T_2 \circ T^*)(P) \sqcup ({T_2}^+ \circ T_1 \circ T^*)(P)$$

Convex hulls are postponed, performed on invariants instead of transformers

⇒ more information is preserved: $P'' \subset P'$

Control-Path Transformers

```
void foo() {
    // $P_0$ : true
    int n = 0; // $T_0 : n' = 0$   ($n'$ : new value of $n$)
    // $P_1 : n = 0$
    while (true) // $T_1 = T_2^* : \text{true}$
        // $P_6 = T_2^*(P_1) : \text{true}$
        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++; // $T_4 : n' \leq 60, n' = n + 1$
            else n = 0; // $T_5 : n > 60, n' = 0$
}
```

Control-Path Transformers

```
void foo() {
    // $P_0$ : true
    int n = 0; // $T_0 : n' = 0$   ($n'$ : new value of $n$)
    // $P_1 : n = 0$
    while (true) // $T_1 = T_2^* :$ true
        // $P_6 = T_2^*(P_1) :$ true
        if (rand()) // $T_2 = T_3 \sqcup \mathrm{Id} : n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
            if (n < 60) n++; // $T_4 : n' \leq 60, n' = n + 1$
            else n = 0; // $T_5 : n > 60, n' = 0$
}
```

With convex hulls in the precondition space:

$$P'_6 = P_1 \sqcup {T_3}^+(P_1) \sqcup \mathrm{Id}^+(P_1) \sqcup (T_3 \circ \mathrm{Id})(P_1) \sqcup (\mathrm{Id} \circ T_3)(P_1) \sqcup \\ ({T_3}^+ \circ \mathrm{Id} \circ (T_3 \sqcup \mathrm{Id}))(P_1) \sqcup (\mathrm{Id}^+ \circ T_3 \circ (T_3 \sqcup \mathrm{Id}))(P_1)$$

Control-Path Transformers

```
void foo() {
    // $P_0$ : true
    int n = 0; // $T_0 : n' = 0$   ($n'$ : new value of $n$)
    // $P_1 : n = 0$
    while (true) // $T_1 = T_2^*$ : true
        // $P_6 = T_2^*(P_1)$ : true
        if (rand()) // $T_2 = T_3 \sqcup \mathrm{Id} : n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
            if (n < 60) n++; // $T_4 : n' \leq 60, n' = n + 1$
            else n = 0; // $T_5 : n > 60, n' = 0$
}
```

With convex hulls in the precondition space:

$$\begin{aligned} P_6' &= P_1 \sqcup {T_3}^+(P_1) \sqcup \mathrm{Id}^+(P_1) \sqcup (T_3 \circ \mathrm{Id})(P_1) \sqcup (\mathrm{Id} \circ T_3)(P_1) \sqcup \\ &\qquad ({T_3}^+ \circ \mathrm{Id} \circ (T_3 \sqcup \mathrm{Id}))(P_1) \sqcup (\mathrm{Id}^+ \circ T_3 \circ (T_3 \sqcup \mathrm{Id}))(P_1) \\ &= P_1 \sqcup {T_3}^+(P_1) \sqcup ({T_3}^+ \circ (T_3 \sqcup \mathrm{Id}))(P_1) \end{aligned}$$

Control-Path Transformers

```
void foo() {
    // $P_0$ : true
    int n = 0; // $T_0 : n' = 0 \quad (n' : \text{new value of } n)$
    // $P_1 : n = 0$
    while (true) // $T_1 = T_2^* : \text{true}$
        // $P_6 = T_2^*(P_1) : \text{true}$
        if (rand()) // $T_2 = T_3 \sqcup \mathsf{Id} : n' \leq n + 1$
            // $T_3 = T_4 \sqcup T_5 : n' \leq 60, n' \leq n + 1$
            if (n < 60) n++; // $T_4 : n' \leq 60, n' = n + 1$
            else n = 0; // $T_5 : n > 60, n' = 0$
}
```

With convex hulls in the precondition space:

$$\begin{aligned} P_6' &= P_1 \sqcup {T_3}^+(P_1) \sqcup \mathsf{Id}^+(P_1) \sqcup (T_3 \circ \mathsf{Id})(P_1) \sqcup (\mathsf{Id} \circ T_3)(P_1) \sqcup \\ &\qquad ({T_3}^+ \circ \mathsf{Id} \circ (T_3 \sqcup \mathsf{Id}))(P_1) \sqcup (\mathsf{Id}^+ \circ T_3 \circ (T_3 \sqcup \mathsf{Id}))(P_1) \\ &= P_1 \sqcup {T_3}^+(P_1) \sqcup ({T_3}^+ \circ (T_3 \sqcup \mathsf{Id}))(P_1) \end{aligned}$$

$$P_6' : 0 \leq n \leq 60$$

Iterative Analysis

- At iteration 1, compute transformers and invariants as usual

[Dillig *et al.*, OOPSLA'13]

```
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]

```
void bar(float x) {
    int i, j = 1, a = 0, b = 0;
    i = 0;
    while (rand()) {
        // P_1 : 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]

```
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]

```
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]

```
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:

Options	None	CP	IA	CP-IA	CP-IA-MP	ASPIC	ISL
Successes	43	69	45	72	73	75	63
Time (s.)	6.1	7.8	18.5	19.6	151.4	10.9	35.5

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

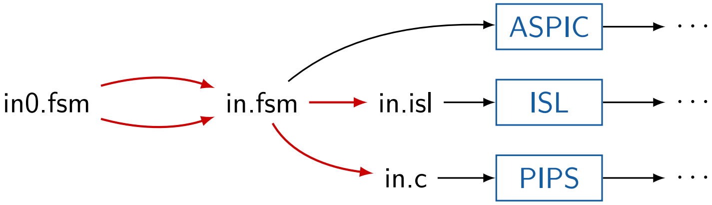
Model Restructuring

Restructure the input model into an equivalent one

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

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

Implemented in ALICe: source-to-source FSM transformation before analysis

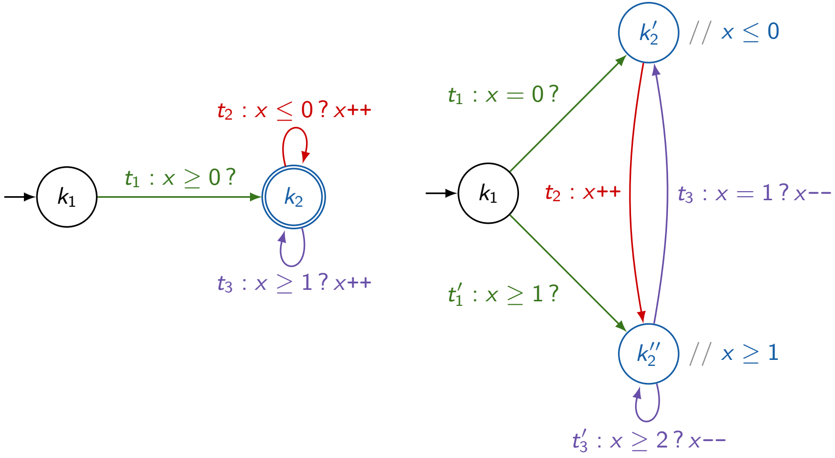


State Splitting

[Maisonneuve NSAD'11]

Designed to get rid of nodes with several self loops, difficult to analyze

Heuristic to split nodes according to loop guards

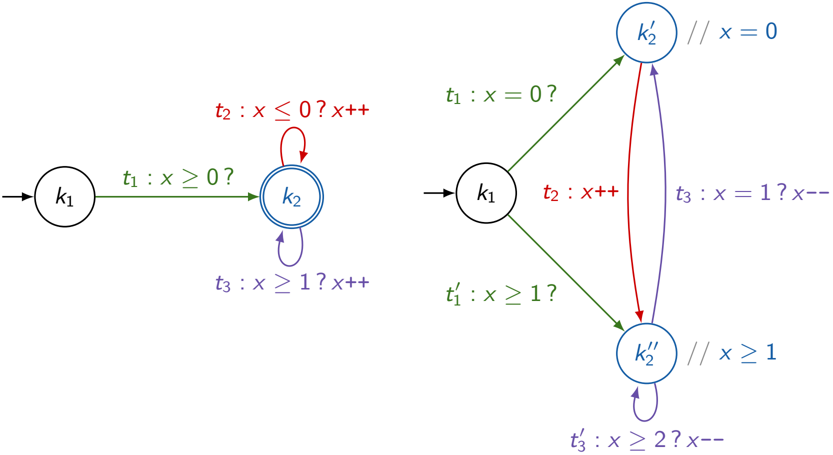


State Splitting

[Maisonneuve NSAD'11]

Designed to get rid of nodes with several self loops, difficult to analyze

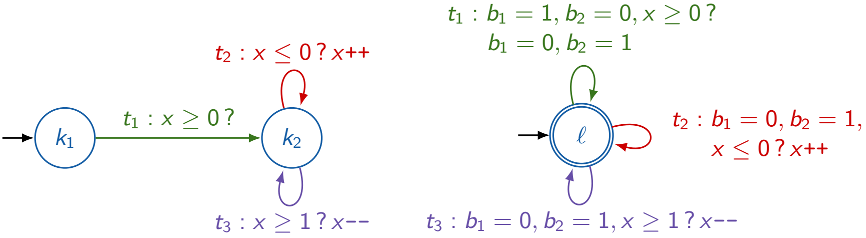
Heuristic to split nodes according to loop guards



State Merge

Transformation to recode the model s.t. it contains only one state 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

Comparative Results

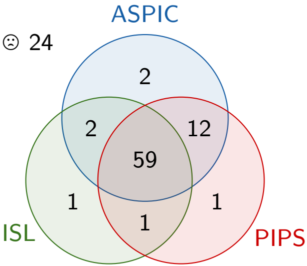
Out of 102 test cases:

	ASPIC	ISL	PIPS
Direct			
Successes	75	63	73
Time (s.)	10.9	35.5	113.2
Split			
Successes	79	72	77
Time (s.)	12.8	43.0	156.3
Merged			
Successes	59	70	68
Time (s.)	16.7	26.2	126.6
Merged + Split			
Successes	70	83	82
Time (s.)	11.3	40.8	146.3

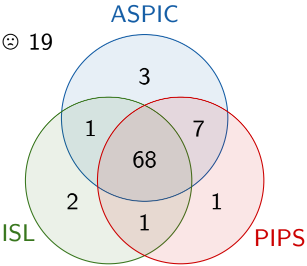
Analysis:

- 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

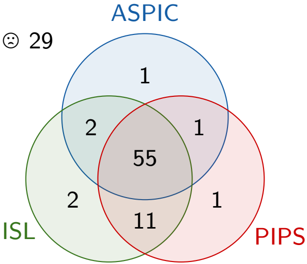
Comparative Results



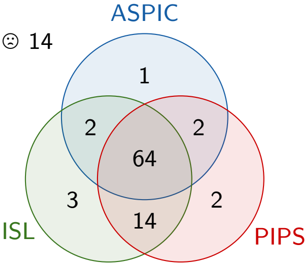
Direct



Split



Merge



Merge + Split

Conclusion on LRA

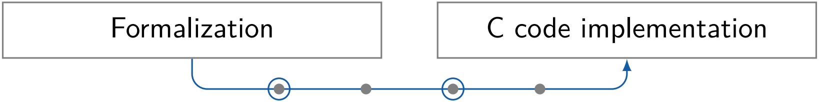
Summary

- 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



Formalization

C code implementation

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

MINES
ParisTech

Thesis defense

Paris, February 6, 2015