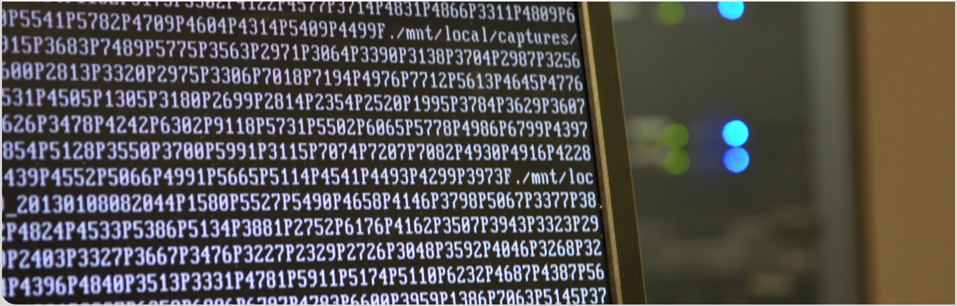
Karlsruher Institut für Technologie

Relational Specification and Verification

From Non-Interference to Regression-free Program Evolution

Bernhard Beckert with M. Kirsten, V. Klebanov, M. Ulbrich, A. Weigl | RS3 Practitioner Event



Relational vs. Function

Functional Verification:

Prove property for one program

Relational Verification:

Prove relation between two programs

Verification of Relational Properties

Karlsruher Institut für Technologie

Use Cases:

- Non-interference / Information flow
- Regression Verification
- Relational Properties of Algorithms
- Refinement

Verification of Relational Properties

Use Cases:

- **Non-interference / Information flow**

$$low_1 = low_2 \;\rightarrow\; \langle P_1 ; P_2 \rangle \; low_1 = low_2$$

- **Regression Verification**

- **Relational Properties of Algorithms**

- **Refinement**

Verification of Relational Properties

Use Cases:

- **Non-interference / Information flow**

$$low_1 = low_2 \;\rightarrow\; \langle P_1 ; P_2 \rangle \; low_1 = low_2$$

- **Regression Verification**

$$in_P = in_Q \;\rightarrow\; \langle P ; Q \rangle \; out_P = out_Q$$

- **Relational Properties of Algorithms**

- **Refinement**

Verification of Relational Properties

Use Cases:

- **Non-interference / Information flow**

$$low_1 = low_2 \;\rightarrow\; \langle P_1;P_2 \rangle \; low_1 = low_2$$

- **Regression Verification**

$$in_P = in_Q \;\rightarrow\; \langle P;Q \rangle \; out_P = out_Q$$

- **Relational Properties of Algorithms**

$$ballots_1 \sim ballots_2 \;\rightarrow\; \langle P_1;P_2 \rangle \; winner_1 \approx winner_2$$

- **Refinement**

Verification of Relational Properties

Use Cases:

- **Non-interference / Information flow**

$$low_1 = low_2 \;\rightarrow\; \langle P_1 ; P_2 \rangle \; low_1 = low_2$$

- **Regression Verification**

$$in_P = in_Q \;\rightarrow\; \langle P ; Q \rangle \; out_P = out_Q$$

- **Relational Properties of Algorithms**

$$ballots_1 \sim ballots_2 \;\rightarrow\; \langle P_1 ; P_2 \rangle \; winner_1 \approx winner_2$$

- **Refinement**

$$in_{Abs} \sim in_{Concr} \;\rightarrow\; \langle Abs ; Concr \rangle \; out_{Abs} \approx out_{Concr}$$

Relational vs. Function

Karlsruher Institut für Technologie

Functional Verification:

Prove property for one program P

Relational Verification:

Prove relation between two programs P, Q

Relational vs. Function

Functional Verification:

Prove property for one program P

Effort grows with size/complexity of P

Relational Verification:

Prove relation between two programs P, Q

Relational vs. Function

Karlsruher Institut für Technologie

Functional Verification:

Prove property for one program P

Effort grows with size/complexity of P

Relational Verification:

Prove relation between two programs P, Q

Effort grows with size/complexity of $\Delta(P, Q)$

Relational vs. Function

Karlsruher Institut für Technologie

Functional Verification:

Prove property for one program P

Effort grows with size/complexity of P

Relational Verification:

Prove relation between two programs P, Q

Effort grows with size/complexity of $\Delta(P, Q)$

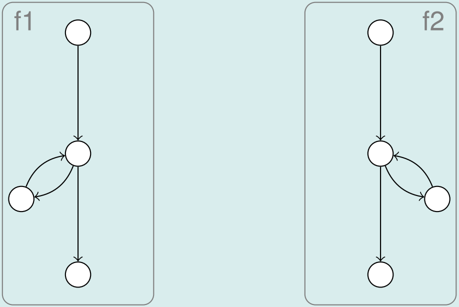
Verification considers P, Q simultaneously!

General Setting

- deductive reasoning
- about complex interferences / flows
- with high precision
- at program level
- “small” programs

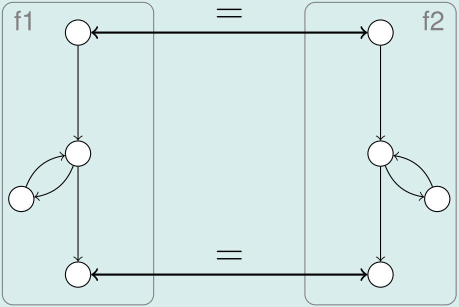
Relational Verification

Loop synchronisation



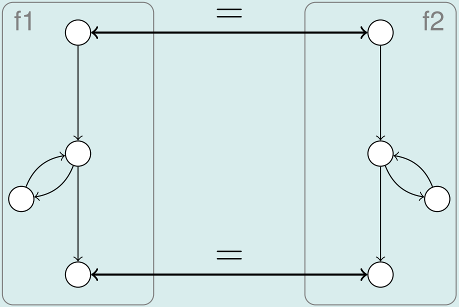
Relational Verification

Loop synchronisation



Relational Verification

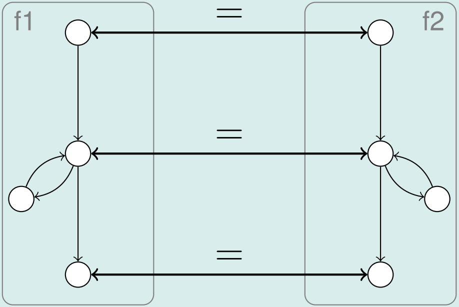
Loop synchronisation



Relational Verification

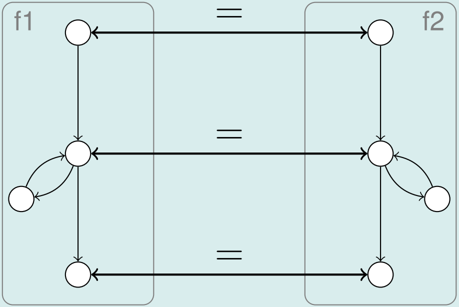
Karlsruher Institut für Technologie

Loop synchronisation



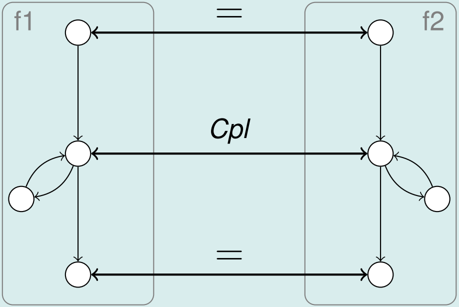
Relational Verification

Loop synchronisation



Relational Verification

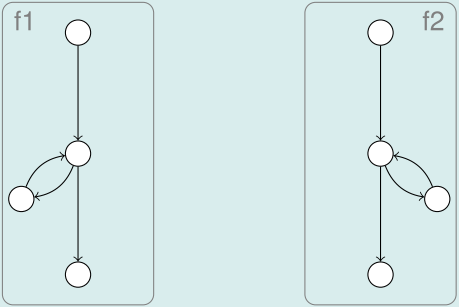
Loop synchronisation



Coupling Invariant ***Cpl***

Relational Verification

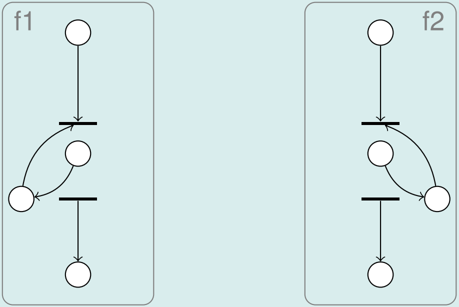
Loop synchronisation



Coupling Invariant ***Cpl***

Relational Verification

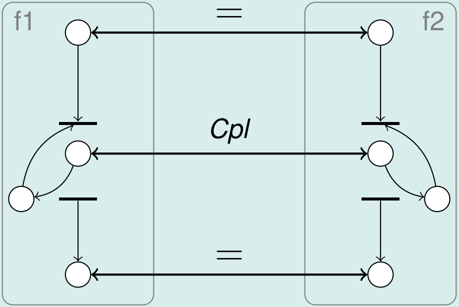
Loop synchronisation



Coupling Invariant *Cpl*

Relational Verification

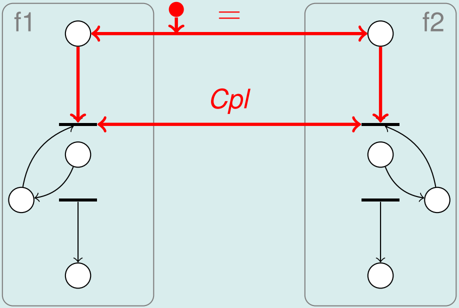
Loop synchronisation



Coupling Invariant *Cpl*

Relational Verification

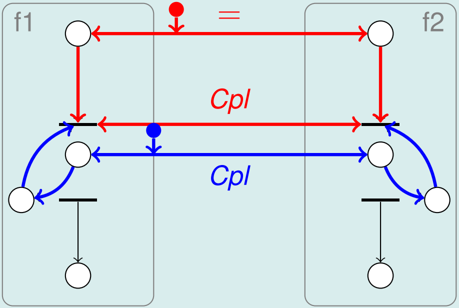
Loop synchronisation



Coupling Invariant ***Cpl***

Relational Verification

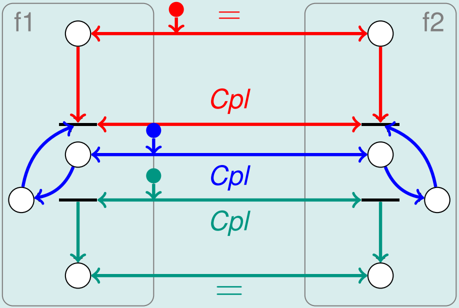
Loop synchronisation



Coupling Invariant ***Cpl***

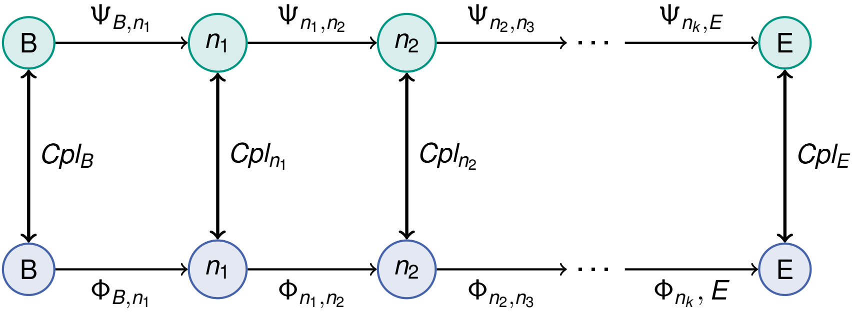
Relational Verification

Loop synchronisation



Coupling Invariant *Cpl*

Synchronised Traces



Relational Verification

for

Object-oriented Programs

KeY Project

www.key-project.org

Project Consortium

- Bernhard Beckert
  Karlsruhe Institute of Technology
- Reiner Hähnle
  TU Darmstadt
- Wolfgang Ahrendt
  Chalmers Univ., Gothenburg

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Project

Karlsruher Institut für Technologie

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

KeY Project

www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)

KeY Project

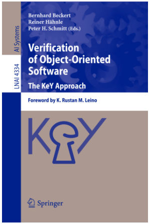
www.key-project.org

Deductive Verification of

- Java
- Specification:
  Java Modeling Language
- Source-code level

KeY Tool

- Deductive rules for all Java features
- Symbolic execution
- Java Card (Java 1.4)
- Semi-automated
  (automation and usability both important)



Object-Sensitive Non-interference

Leakage by aliasing

```
void m() {
    C c1 = new C();    // new obj
    C c2 = c1;         // alias
    c2.x = high;
    low  = c1.x;
}
```

Karlsruher Institut für Technologie

Leakage by aliasing

```
void m() {
   C c1 = new C();    // new obj
   C c2 = c1;         // alias
   c2.x = high;
   low  = c1.x;
}
```

NOT SECURE

Object-Sensitive Non-interference

Object creation and object identity

```
if (high>0) {
    low1 = new C();
    low2 = new C();
} else {
    low2 = new C();
    low1 = new C();
}
```

Object-Sensitive Non-interference

Object creation and object identity

```
if (high>0) {
    low1 = new C();
    low2 = new C();
} else {
    low2 = new C();
    low1 = new C();
}
```

SECURE

E-Voting Case Study

Joint work with

- Ralph Küsters, Trier
- Gregor Snelting, Karlsruhe

E-Voting Case Study

Joint work with

- Ralph Küsters, Trier
- Gregor Snelting, Karlsruhe

Proof Goal

The sum is the only information about the votes that is leaked.

E-Voting Case Study

Karlsruher Institut für Technologie

Joint work with

- Ralph Küsters, Trier
- Gregor Snelting, Karlsruhe

Proof Goal

The sum is the only information about the votes that is leaked.

Hybrid Approach [GRSD 2013]

information-flow analysis in JOANA (w/o declassification)
$+$ functional verification in KeY
$=$ non-interference with declassification

E-Voting Case Study

Joint work with

- Ralph Küsters, Trier
- Gregor Snelting, Karlsruhe

Proof Goal

The sum is the only information about the votes that is leaked.

Hybrid Approach [GRSD 2013]

information-flow analysis in JOANA (w/o declassification)
$+$ functional verification in KeY
$=$ non-interference with declassification

Simplified system fully verified (functional and information-flow)

Relational Verification

of

Programmable Logic Controllers

PLC Software Equivalence

PLC Software Equivalence

Collaboration with

TUM

Technische Universität München

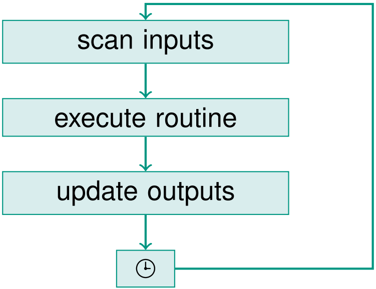
Prof. Vogel-Heuser

Programmable Logic Controllers (PLCs)

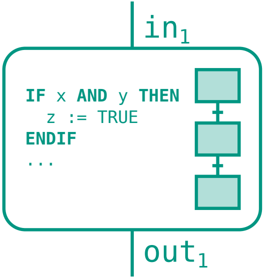
- special-purpose programming languages (IEC 61131, . . .)
- simple structure (scan cycles)

Programmable Logic Controllers (PLCs)

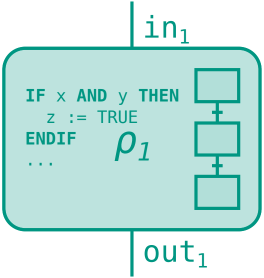
- special-purpose programming languages (IEC 61131, . . .)
- simple structure (scan cycles)



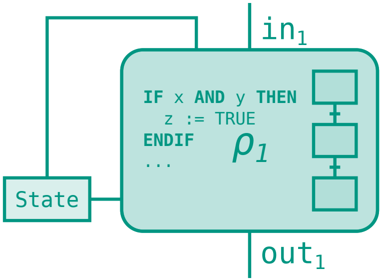
Equivalence of PLC Programs



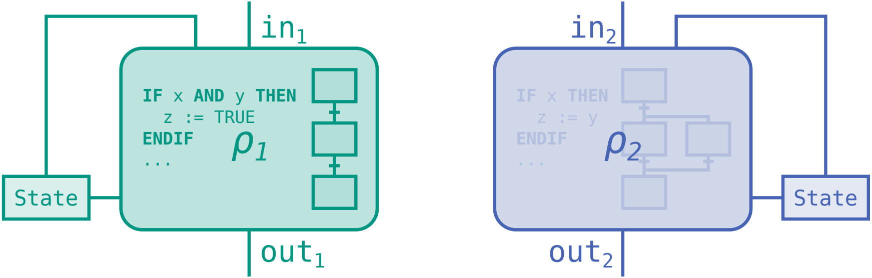
Equivalence of PLC Programs



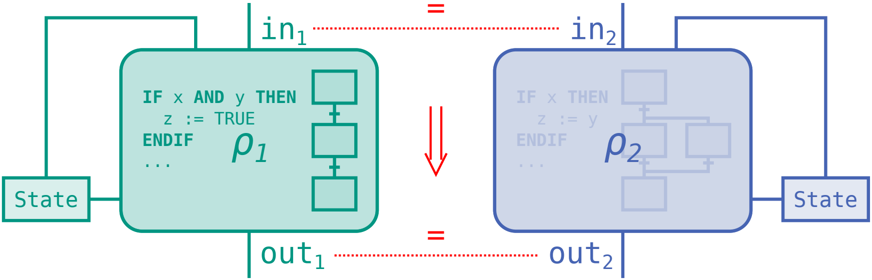
Equivalence of PLC Programs



Equivalence of PLC Programs

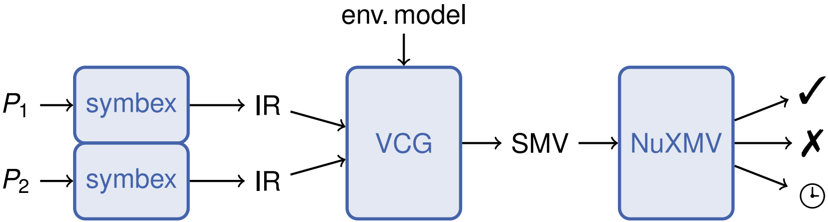


Equivalence of PLC Programs



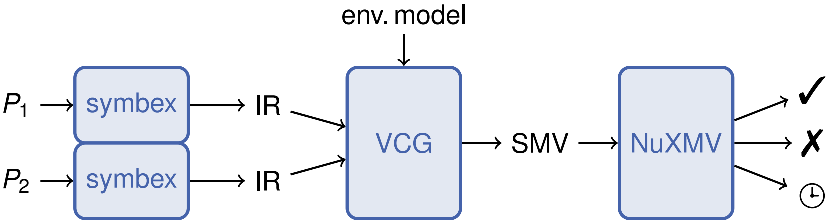
Workflow

Relational Verfication Workflow for PLC:



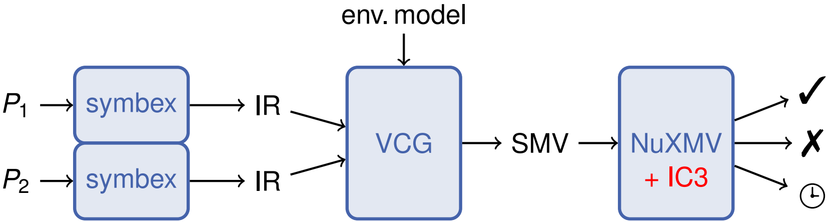
Workflow

Relational Verfication Workflow for PLC:



Workflow

Relational Verfication Workflow for PLC:



Attacker Model

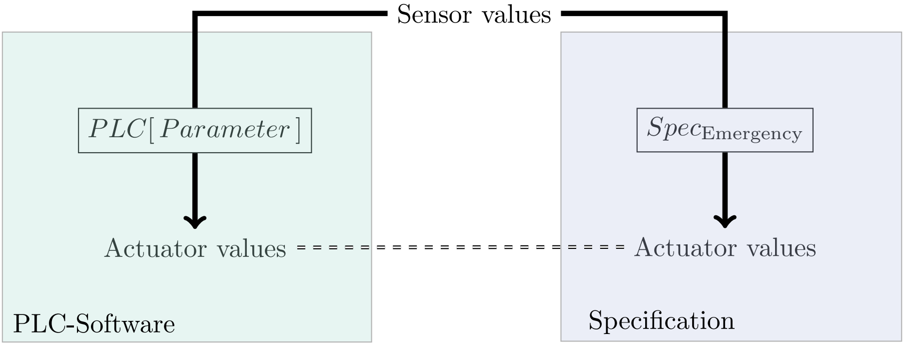
Attacker: Can change system parameters (remote maintenace)

Attacker Model

Attacker: Can change system parameters (remote maintenace)

Proof Goal: No interference with critical functionality (safety features)

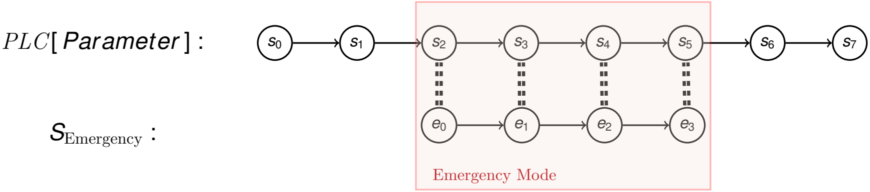
Non-interference of Parameters with Safety Feature



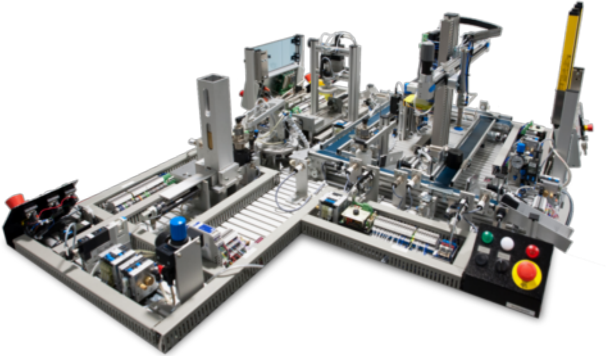
Non-interference of Parameters with Safety Feature

Proof Obligation

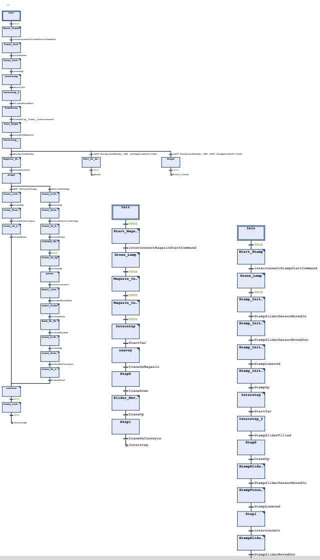
$$\forall Parameter: \ \mathbf{G}\,(Emergency \rightarrow PLC[\,Parameter\,] \approx Spec_{\mathrm{Emergency}})$$

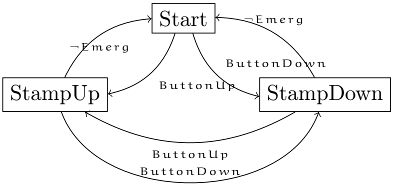


Case Study: Pick-and-Place Unit

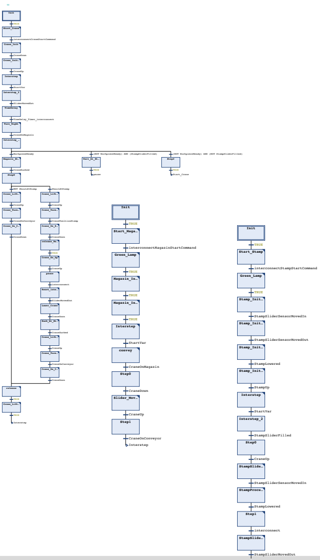


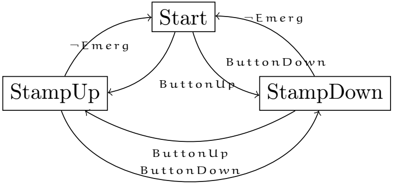
Case Study: Pick-and-Place Unit





Case Study: Pick-and-Place Unit





Relational: 300ms

Functional: timeout (30 min)

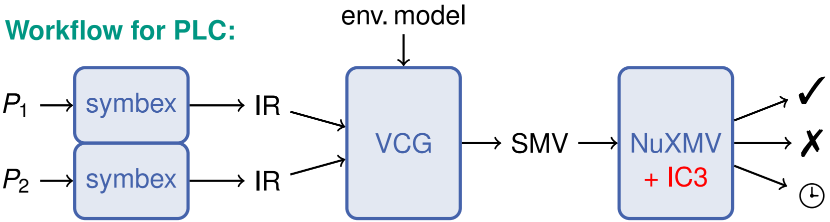
Relational Verification

of

C Programs

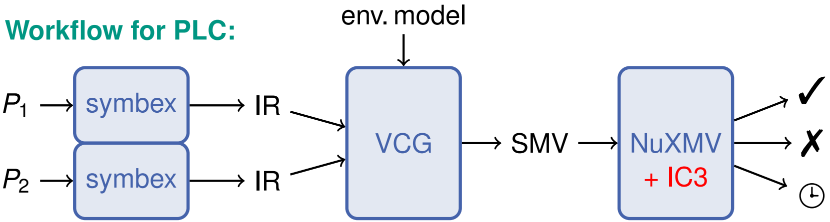
Workflow

Workflow for PLC:

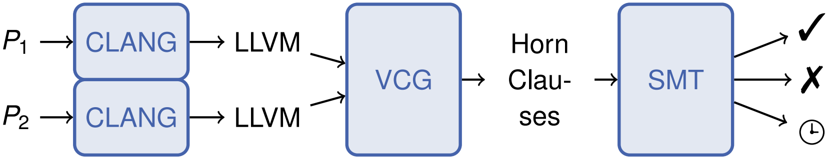


Workflow

Workflow for PLC:



Workflow for C Programs:



Coupling invariant inferred automatically!

(Z3/Eldarica)

Our approach by example

Karlsruher Institut für Technologie

Original code

```
int f1(int n) {
  int result = 1;
  n = n/10;
  while(n > 0) {
    result ++;
    n = n/10;
  }
  return result;
}
```

What does it compute?

Our approach by example

Karlsruher Institut für Technologie

Original code

```
int f1(int n) {
  int result = 1;
  n = n/10;
  while(n > 0) {
    result ++;
    n = n/10;
  }
  return result;
}
```

What does it compute?

... the number of decimal digits of a non-negative number *n*.

Behaviour preserved?

Original code

```
int f1(int n) {
  int result = 1;
  n = n/10;
  while(n > 0) {
    result ++;
    n = n/10;
  }
  return result;
}
```

Optimised version

```
int f2(int n) {
  int result = 1;
  while(true) {
    if(n<10) return result;
    if(n<100) return result+1;
    if(n<1000) return result+2;
    if(n<10000) return result+3;
    n /= 10000; result += 4;
  }
  return result;
}
```

Optimisation uses fewer divisions ($\approx$ 7 times faster)

[A. Alexandrescu. *Three optimization tips for C++*, 2012]

Demo:
rêve Tool

Regression verification:

`formal.iti.kit.edu/improve/reve/`

Non-interference:

`formal.iti.kit.edu/improve/reve/noninter/`

Try for yourself!

llrêve

[Usage | Disclaimer]

Automatically check two programs for equivalence

Developed and maintained as part of the IMPROVE project by Mattias Ulbrich, Vladimir Klebanov and Moritz Kiefer.

Load a predefined example:

memset_1 (examples suffixed with ! contain programs that do **not** behave equally)

or enter two programs:

```
/* openbsd */
#include <stddef.h>
extern int __mark(int);

void *
memset(void *dst, int c, size_t n)
{
	if (n != 0) {
		unsigned char *d = dst;

		do
			*d++ = (unsigned char)c;
		while (__mark(0) & (--n != 0));
	}
	return (dst);  }
```

```
/* dietlibc */
#include <stddef.h>
extern int __mark(int);
void* memset(void * dst, int s, size_t count) {
    register char * a = dst;
    count++;    /* this actually creates smaller code than using count-- */
    while (--count) {
        *a++ = s;
        __mark(0);
    }
    return dst;
}
```

Check equivalence Your programs are sent to the server. Please be a little patient for the answer...

› **Running llrêve**

› **Running Eldarica**

PROGRAMS HAVE BEEN PROVED EQUIVALENT.

```
sat
inv_main_0(A,B,C,D,E,F,G,H,I,J,K,L) :- (((((A = G), (B = J)), (C = H)), (D = I)), (\+((E = K)); (F =
L))).
```

sat means that the two programs behave equally.

unsat means that there is at least one input on which the programs behave differently. See counterexample below.

If no verdict is presented, the tool may have timed out (TO set to 60s for this server)

llrêve – non-interference –

[Usage | Disclaimer]

Automatically check non-interference for C programs

Developed and maintained as part of the IMPROVE project by Mattias Ulbrich, Vladimir Klebanov and Moritz Kiefer.

Load a predefined example:

hammer (examples suffixed with ! contain programs that do **not** behave equally)

or enter a program:

Parameters which are "low":

Program:

```
int secure_ifl(int high) {
  int x = 0;
  int y = 0;
  int sum = 0;
  while (__mark(1) & y < 10) {
    sum += x;
    if (y == 5) {
      x = high;
      y = 9;
    }
    x++;
    y++;
  }
  return sum;     // This is a secure example from a tutorial by Christian Hammer
}
```

Check non-interference Your program is sent to the server. Please be a little patient for the answer ...

Program proved non-interferent: No flow from high to return value.

```
sat
inv_main_1(A,B,C,D,E,F,G,H) :- (((((((((B = F), (C = G)), (D = 5)), (H = 5)); (((((B =
F), (C = 4)), (D = 4)), (G = 4)), (H = 4))); (((((B = F), (C = 3)), (D = 3)), (G = 3)),
(H = 3))); (((((B = F), (C = 2)), (D = 2)), (G = 2)), (H = 2))); (((((B = F), (C = 1)),
(D = 1)), (G = 1)), (H = 1))); (((((B = F), (C = 0)), (D = 0)), (G = 0)), (H = 0)));
((((B = F), (H = 10)), (D >= 10)), (((-8 - D) mod 6) = 0))).
```