Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices

Jean-Christophe Léchenet, Nikolai Kosmatov, Pascale Le Gall

AFADL 2016, Besançon, 7 June 2016

Definition

Static backward slicing (introduced by Weiser in 1981)

- simplifies a given program p but preserves the behavior w.r.t. a point of interest C (slicing criterion, typically a statement)
- removes irrelevant statements that do not impact C
- produces a simplified program q (slice)

Example: a program and a slice

Check if a is divisible by b.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

Example: a program and a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

euclidian division of *a* by *b*

is the remainder equal to 0 ?

Original program *p*

Example: a program and a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice *q* w.r.t. line 8

- Goal: preserve the behaviour of *p* w.r.t. line 8

Global motivation

- Traditional scope of slicing
  - program understanding
  - debugging : understand an already found error
    [Weiser, 1982] [Agrawal et al., 1993] [Hierons et al., 1999]
- Frama-C: an extensible platform for analysis of C code
  - ACSL annotation language
  - Plugins for value analysis, proof, testing, slicing...
  - `http://frama-c.com/`

Our purpose

Conduct V&V on a slice instead of the initial program.

Dependence-based slicing

- WHILE language: `skip`, `x:=e`, `if`, `while`

Control dependence

```
if (l: b) {
    ...
    l_then: stmt;
    ...
} else {
    ...
    l_else: stmt;
    ...
}
```

```
while (l: b) {
    ...
    l_body: stmt;
    ...
}
```

Data dependence

```
l_def: x = e; // def
... // x not assigned
... // x not assigned
... // x not assigned
l_use: y = ... x ...; // use
```

- Dependence-based slice q of p w.r.t. C: all statements on which one of the statements of C is (directly or indirectly) dependent
- Formally: q $= \{l \in p \mid l \rightarrow^* l', l' \in C\}$, where $\rightarrow = \xrightarrow{ctrl} \cup \xrightarrow{data}$

Example: computing a slice

Check if a is divisible by b.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

?

control

data

Slice q w.r.t. line 8

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

?

control

data

Slice *q* w.r.t. line 8

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

?

Slice *q* w.r.t. line 8

control

data

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

?

Slice *q* w.r.t. line 8

control

data

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

?

Slice *q* w.r.t. line 8

control

data

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

?

Slice *q* w.r.t. line 8

control

data

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

?

Slice *q* w.r.t. line 8

control

data

Example: computing a slice

Check if *a* is divisible by *b*.

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program *p*

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice *q* w.r.t. line 8

control

data

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{l} \quad (label, \quad (q, \quad r, \quad res \quad)) \\ \mathcal{T}[\![p]\!]\sigma = \langle \\ \\ \quad \rangle \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{ll} & (label, \quad (q, \quad r, \quad res \quad)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, \qquad (0, \quad 0, \quad 0 \quad)) \\ \rangle & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rlllll} & (label, & (q, & r, & res &)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\ & (2, & (0, & 2, & 0 &)) \\ & & & & & \\ \rangle & & & & & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0,\\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rlllll} & (label, & (q, & r, & res &)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\ & (2, & (0, & 2, & 0 &)) \\ & (3, & (0, & 2, & 0 &)) \\ & & & & & \\ \rangle & & & & & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rlllll} & (label, & (q, & r, & res &)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\ & (2, & (0, & 2, & 0 &)) \\ & (3, & (0, & 2, & 0 &)) \\ & (4, & (1, & 2, & 0 &)) \\ & \rangle & & & & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rllll} & (label, & (q, & r, & res \quad)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 \quad)) \\ & (2, & (0, & 2, & 0 \quad)) \\ & (3, & (0, & 2, & 0 \quad)) \\ & (4, & (1, & 2, & 0 \quad)) \\ & (5, & (1, & 0, & 0 \quad)) \\ & \rangle & & & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\}:$$

The trajectory of p on σ is:

$$\begin{array}{rlllll} & (label, & (q, & r, & res &)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\ & (2, & (0, & 2, & 0 &)) \\ & (3, & (0, & 2, & 0 &)) \\ & (4, & (1, & 2, & 0 &)) \\ & (5, & (1, & 0, & 0 &)) \\ & (3, & (1, & 0, & 0 &)) \\ & \rangle & & & & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0,\\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rlllll} & (label, & (q, & r, & res &)) \\ \mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\ & (2, & (0, & 2, & 0 &)) \\ & (3, & (0, & 2, & 0 &)) \\ & (4, & (1, & 2, & 0 &)) \\ & (5, & (1, & 0, & 0 &)) \\ & (3, & (1, & 0, & 0 &)) \\ & (6, & (1, & 0, & 0 &)) \\ & \rangle & & & & \end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rlllll}
 & (label, & (q, & r, & res &)) \\
\mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\
 & (2, & (0, & 2, & 0 &)) \\
 & (3, & (0, & 2, & 0 &)) \\
 & (4, & (1, & 2, & 0 &)) \\
 & (5, & (1, & 0, & 0 &)) \\
 & (3, & (1, & 0, & 0 &)) \\
 & (6, & (1, & 0, & 0 &)) \\
 & (8, & (1, & 0, & 1 &)) \\
 & \rangle & & & &
\end{array}$$

Introduction to trajectories [Weiser, 1981]

```
1 : q = 0;
2 : r = a;
3 : while (b <= r) {
4 :     q = q + 1;
5 :     r = r - b;
    }
6 : if (r != 0) {
7 :     res = 0;
    } else {
8 :     res = 1;
    }
```

Original program p

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0,\\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of p on σ is:

$$\begin{array}{rlllll}
 & (label, & (q, & r, & res &)) \\
\mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\
 & (2, & (0, & 2, & 0 &)) \\
 & (3, & (0, & 2, & 0 &)) \\
 & (4, & (1, & 2, & 0 &)) \\
 & (5, & (1, & 0, & 0 &)) \\
 & (3, & (1, & 0, & 0 &)) \\
 & (6, & (1, & 0, & 0 &)) \\
 & (8, & (1, & 0, & 1 &)) \\
 & \rangle & & & &
\end{array}$$

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\mathcal{T}[\![q]\!]\sigma = \langle \quad (label, \quad (r, \quad res \quad)) \quad \rangle$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0,$$
$$r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{lllll} & (label, & (r, & res &)) \\ \mathcal{T}[\![q]\!]\sigma = \langle & (2, & (2, & 0 &)) \\ \rangle & & & & \end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{llll} & (label, & (r, & res \quad)) \\ \mathcal{T}[\![q]\!]\sigma = \langle & ((2, & (2, & 0 \quad)) \\ & (3, & (2, & 0 \quad)) \\ & & & \\ \rangle & & & \end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{llll} & (label, & (r, & res &)) \\ \mathcal{T}[\![q]\!]\sigma = \langle & (2, & (2, & 0 &)) \\ & (3, & (2, & 0 &)) \\ & (5, & (0, & 0 &)) \\ \rangle & & & & \end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0,$$
$$r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{llll} & (label, & (r, & res \quad)) \\ \mathcal{T}[\![q]\!]\sigma = \langle & ((2, & (2, & 0 \quad)) \\ & (3, & (2, & 0 \quad)) \\ & (5, & (0, & 0 \quad)) \\ & \color{red}{(3,} & \color{red}{(0,} & \color{red}{0} \quad)) \\ & \rangle & & \end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{lllll} & (label, & (r, & res &)) \\ \mathcal{T}[\![q]\!]\sigma = \langle & (2, & (2, & 0 &)) \\ & (3, & (2, & 0 &)) \\ & (5, & (0, & 0 &)) \\ & (3, & (0, & 0 &)) \\ & (6, & (0, & 0 &)) \\ & \rangle & & & \end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {


    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0, \\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{lllll} & (label, & (r, & res &)) \\ \mathcal{T}[\![q]\!]\sigma = \langle & (2, & (2, & 0 &)) \\ & (3, & (2, & 0 &)) \\ & (5, & (0, & 0 &)) \\ & (3, & (0, & 0 &)) \\ & (6, & (0, & 0 &)) \\ & (8, & (0, & 1 &)) \\ & \rangle & & & \end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {

    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to trajectories [Weiser, 1981]

For initial state

$$\sigma = \{a \mapsto 2, b \mapsto 2, q \mapsto 0,\\ r \mapsto 0, res \mapsto 0\} :$$

The trajectory of q on σ is:

$$\begin{array}{rllll}
 & (label, & (r, & res &)) \\
\mathcal{T}[\![q]\!]\sigma = \langle & (2, & (2, & 0 &)) \\
 & (3, & (2, & 0 &)) \\
 & (5, & (0, & 0 &)) \\
 & (3, & (0, & 0 &)) \\
 & (6, & (0, & 0 &)) \\
 & (8, & (0, & 1 &)) \\
 & \rangle & & &
\end{array}$$

```
2 : r = a;
3 : while (b <= r) {

5 :     r = r - b;
    }
6 : if (r != 0) {


    } else {
8 :     res = 1;
    }
```

Slice q w.r.t. line 8

Introduction to projections [Weiser, 1981]

How to compare both trajectories ? They should agree on preserved statements for preserved variables.

$$
\begin{array}{rlllll}
 & (label, & (q, & r, & res &)) \\
\mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\
 & (2, & (0, & 2, & 0 &)) \\
 & (3, & (0, & 2, & 0 &)) \\
 & (4, & (1, & 2, & 0 &)) \\
 & (5, & (1, & 0, & 0 &)) \\
 & (3, & (1, & 0, & 0 &)) \\
 & (6, & (1, & 0, & 0 &)) \\
 & (8, & (1, & 0, & 1 &)) \\
 & \rangle & & & &
\end{array}
\qquad
\begin{array}{rllll}
 & (label, & (r, & res &)) \\
\mathcal{T}[\![q]\!]\sigma = \langle & & & & \\
 & (2, & (2, & 0 &)) \\
 & (3, & (2, & 0 &)) \\
 & & & & \\
 & (5, & (0, & 0 &)) \\
 & (3, & (0, & 0 &)) \\
 & (6, & (0, & 0 &)) \\
 & (8, & (0, & 1 &)) \\
 & \rangle & & &
\end{array}
$$

Introduction to projections [Weiser, 1981]

How to compare both trajectories ? They should agree on preserved statements for preserved variables.

$$
\begin{array}{rlllll}
 & (label, & (q, & r, & res &)) \\
\mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\
 & (2, & (0, & 2, & 0 &)) \\
 & (3, & (0, & 2, & 0 &)) \\
 & (4, & (1, & 2, & 0 &)) \\
 & (5, & (1, & 0, & 0 &)) \\
 & (3, & (1, & 0, & 0 &)) \\
 & (6, & (1, & 0, & 0 &)) \\
 & (8, & (1, & 0, & 1 &)) \\
 & \rangle & & & &
\end{array}
\qquad
\begin{array}{rllll}
 & (label, & (r, & res &)) \\
\mathcal{T}[\![q]\!]\sigma = \langle & & & & \\
 & (2, & (2, & 0 &)) \\
 & (3, & (2, & 0 &)) \\
 & & & & \\
 & (5, & (0, & 0 &)) \\
 & (3, & (0, & 0 &)) \\
 & (6, & (0, & 0 &)) \\
 & (8, & (0, & 1 &)) \\
 & \rangle & & &
\end{array}
$$

Introduction to projections [Weiser, 1981]

How to compare both trajectories ? They should agree on preserved statements for preserved variables.

$$
\begin{array}{rlllll}
 & (label, & (q, & r, & res &)) \\
\mathcal{T}[\![p]\!]\sigma = \langle & (1, & (0, & 0, & 0 &)) \\
 & (2, & (0, & 2, & 0 &)) \\
 & (3, & (0, & 2, & 0 &)) \\
 & (4, & (1, & 2, & 0 &)) \\
 & (5, & (1, & 0, & 0 &)) \\
 & (3, & (1, & 0, & 0 &)) \\
 & (6, & (1, & 0, & 0 &)) \\
 & (8, & (1, & 0, & 1 &)) \\
 & \rangle & & & &
\end{array}
\qquad
\begin{array}{rllll}
 & (label, & (r, & res &)) \\
\mathcal{T}[\![q]\!]\sigma = \langle & & & & \\
 & (2, & (2, & 0 &)) \\
 & (3, & (2, & 0 &)) \\
 & & & & \\
 & (5, & (0, & 0 &)) \\
 & (3, & (0, & 0 &)) \\
 & (6, & (0, & 0 &)) \\
 & (8, & (0, & 1 &)) \\
 & \rangle & & &
\end{array}
$$

Classic soundness property

Let q be a slice of p.

Theorem (Classic soundness property, [Weiser, 1981])

Let σ be an input state of p. Suppose that p halts on σ. Then q halts on σ and the executions of p and q on σ agree after each statement preserved in the slice on the variables that appear in this statement.

- Formalized with a trajectory-based semantics as an equality of projections: $Proj_{L(q)}(\mathcal{T}[\![p]\!]\sigma) = Proj_{L(q)}(\mathcal{T}[\![q]\!]\sigma)$

Application to V&V

Does this result hold in the general case, i.e. in presence of errors and non-termination ?

Assertions to model errors

- WHILE language: `skip`, `x:=e`, `if`, `while`, `assert`
- Assertions make runtime errors explicit
- Assertions protect all statements that may cause a runtime error

```
assert (l: N != 0);
l1: x = k/N;
```

```
assert (l: k < N);
l1: x = a[k];
```

Case 1: same error

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Original program p

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Slice q w.r.t. line 20

Execution for test input: $N = 2$, $k = 4$

Case 2: error hidden by another error (not preserved)

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Original program p

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Slice q w.r.t. line 18

Execution for test input: $N = 0$, $k = 0$

Case 3: error hidden by a loop (not preserved)

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Original program p

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Slice q w.r.t. line 20

Execution for test input: $N = 4,\ k = 0$

- Equality of projections does not hold in general, due to:
  - Non-termination [Ball et al.,1993] [Ranganath et al., 2007] [Amtoft, 2008]
  - Errors [Harman et al., 1995]
- Three possible directions:
  - change the semantics [Cartwright et al., 1989] [Giacobazzi et al., 2003] [Nestra, 2009] [Barraclough et al., 2010]
    - ☺ Extend the classic soundness property
    - ☹ Consider non-existing trajectories
  - add more dependencies [Ranganath et al., 2007]
    - ☺ Extend the classic soundness property
    - ☹ Bigger slices
      All loops and assertions preceding the criterion will be systematically preserved
  - keep same kind of dependencies [Amtoft, 2008]
    - ☺ Keep slices small
    - ⚠ Another soundness property required

- Equality of projections does not hold in general, due to:
  - Non-termination [Ball et al.,1993] [Ranganath et al., 2007] [Amtoft, 2008]
  - Errors [Harman et al., 1995]
- Three possible directions:
  - change the semantics [Cartwright et al., 1989] [Giacobazzi et al., 2003] [Nestra, 2009] [Barraclough et al., 2010]
    - ☺ Extend the classic soundness property
    - ☹ Consider non-existing trajectories
  - add more dependencies [Ranganath et al., 2007]
    - ☺ Extend the classic soundness property
    - ☹ Bigger slices
      All loops and assertions preceding the criterion will be systematically preserved
  - **keep same kind of dependencies** [Amtoft, 2008]
    - ☺ Keep slices small
    - ⚠ Another soundness property required

Relaxed dependence-based slicing

- In addition to (unmodified) control and data dependencies, we introduce assertion dependencies between assertions and their protected statements

Assertion dependence

```
assert (l: N != 0);
l1: x = k/N;
```

```
assert (l: k < N);
l1: x = a[k];
```

- Relaxed slice q of p w.r.t. C: all statements on which one of the statements of C is (directly or indirectly) dependent
- Formally: $q = \{l \in p \mid l \to^* l', l' \in C\}$, where $\to = \xrightarrow{ctrl} \cup \xrightarrow{data} \cup \xrightarrow{assert}$

Coq proof assistant

- An interactive theorem prover
- Developed since 1984 by Inria
- Extraction of certified Coq (OCaml, Haskell)
- Used for: the four color theorem, CompCert...

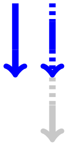
`https://coq.inria.fr/`

Soundness property of relaxed slicing

Let q be a slice of p.

Theorem

The projection of the trajectory of p is a prefix of the projection of the trajectory of q. If the execution of p terminates normally, the projections are equal.



Corollary

The classic soundness property.

Case 2: error hidden by another error (not preserved)

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Original program *p*

```
1  s1 = 0;
2  s2 = 0;
3  i = 0;
4  while (i < N){
5    assert (i < N);
6    s1 = s1 + a[i];
7    i = i + k;
8  }
9  j = 0;
10 assert (k != 0);
11 last = N/k;
12 while (j <= last){
13   assert (k*j < N);
14   s2 = s2 + a[k*j];
15   j = j + 1;
16 }
17 assert (N != 0);
18 avg1 = s1 / N;
19 assert (N != 0);
20 avg2 = s2 / N;
21 if(avg1 == avg2)
22   print("equal");
```

Slice *q* w.r.t. line 18

Execution for test input: $N = 0$, $k = 0$

Verification on relaxed slices

Let q be a slice of p.

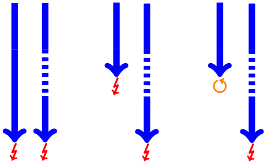
Theorem (No errors in the slice)

If there are no runtime errors in q, then there are none in p, in the statements preserved in q.

```
s1 = 0;
s2 = 0;
i = 0;
while (i < N){
  assert (i < N);
  s1 = s1 + a[i];
  i = i + k;
}
j = 0;
assert (k != 0);
last = N/k;
while (j <= last){
  assert (k*j < N);
  s2 = s2 + a[k*j];
  j = j + 1;
}
...
```

Theorem (An error in the slice)

If there is a runtime error in q, then either the same error occurs in p, or another error or an infinite loop caused by a statement not preserved in q masks it.



Technical point

Control dependence

```
if (l: b) {
    ...
    l_then: stmt;
    ...
} else {
    ...
    l_else: stmt;
    ...
}
```

```
while (l: b) {
    ...
    l_body: stmt;
    ...
}
```

Data dependence

```
l_def: x = e; // def
... // x not assigned
... // x not assigned
... // x not assigned
l_use: y = ... x ...; // use
```

Problems with data dependence:

- Loops give an infinite number of unbounded executions
- Not induction-friendly
  - How to compute the data dependencies of the sequence of two programs, given the data dependencies of each one ?

Our solution:

- Reformulating data dependence in an induction-friendly way

Reformulation of data dependence for $(p_1; p_2)$

```
l1: x = e; // def
... // x not assigned
l2: y = ... x...; // use
... // y not assigned
```

$\left.\right\} p_1$

Reformulation of data dependence for $(p_1; p_2)$

$$
\left.\begin{array}{l}
\texttt{... // y not assigned} \\
\texttt{l3: z = ... y ...; //def} \\
\texttt{... // z not assigned} \\
\texttt{l4: a = ... z ...; // use}
\end{array}\right\} p_2
$$

Reformulation of data dependence for $(p_1; p_2)$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned
```

$\Big\}\ p_1$

```
... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

$\Big\}\ p_2$

Reformulation of data dependence for $(p_1; p_2)$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned
```

p_1

```
... // y not assigned
l3: z = ... y...; // def
... // z not assigned
l4: a = ... z ...; //use
```

p_2

Reformulation of data dependence for $(p_1; p_2)$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned

... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_1

p_2

$$data(p1; p2) = ?$$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned

... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_1

p_2

$$data(p1; p2) = \mathbf{data(p1)} \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

p_1

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned
```

```
... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

def-use paths inside $p2$

p_2

$$data(p1; p2) = data(p1) \cup \mathbf{data(p2)} \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

def-use paths from $p1$ to $p2$

def-use paths inside $p2$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned

... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_1

p_2

$$data(p1; p2) = data(p1) \cup data(p2) \ \cup \ \mathbf{?}$$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned
```

p_1

```
... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_2

def-use paths inside $p2$

$$data(p1; p2) = data(p1) \cup data(p2) \cup ?$$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned
```

p_1

$maydef(p_1)$

```
... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_2

def-use paths inside $p2$

$$maydef(p) = \{\, (\mathtt{l}, v) \,|\, \mathtt{l} \text{ can be the last definition of } v \text{ in } p \,\}$$

$$data(p1; p2) = data(p1) \cup data(p2) \;\cup\; ?$$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

$maydef(p_1)$

$mayuse(p_2)$

def-use paths inside $p2$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned

... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_1

p_2

$maydef(p) = \{ (\mathtt{l}, v) \mid \mathtt{l} \text{ can be the last definition of } v \text{ in } p \}$
$mayuse(p) = \{ (\mathtt{l}, v) \mid v \text{ can be read at } \mathtt{l} \text{ before being redefined } \}$
$data(p1; p2) = data(p1) \cup data(p2) \ \cup \ ?$

Reformulation of data dependence for $(p_1; p_2)$

def-use paths inside $p1$

$maydef(p_1)$

$mayuse(p_2)$

def-use paths inside $p2$

```
l1: x = e; //def
... // x not assigned
l2: y = ... x ...; //use
... // y not assigned

... // y not assigned
l3: z = ... y ...; //def
... // z not assigned
l4: a = ... z ...; //use
```

p_1

p_2

$$maydef(p) = \{ (\mathtt{l}, v) \mid \mathtt{l} \text{ can be the last definition of } v \text{ in } p \}$$

$$mayuse(p) = \{ (\mathtt{l}, v) \mid v \text{ can be read at } \mathtt{l} \text{ before being redefined } \}$$

$$\begin{aligned} data(p1; p2) = {} & data(p1) \cup data(p2) \cup \\ & \{ (\mathtt{l}, \mathtt{l'}) \in (p1, p2) \mid \exists v : \begin{array}{ll} (\mathtt{l}, v) & \in maydef(p1) \wedge \\ (\mathtt{l'}, v) & \in mayuse(p2) \} \end{array} \end{aligned}$$

Certified

- Certified intra- and inter-procedural program slicing in Isabelle/HOL
  [Wasserab, 2011]
- A posteriori validation of program slicing integrated in CompCert (in Coq)
  [Blazy et al., 2015]

Unlike the previous works, our purpose was:

- a machine-checked proof of the soundness of slicing in presence of errors and non-termination
- a justification of V&V on slices

Conclusion:

- Relaxed slicing: soundness, yet slices of reasonable size
- A formal link about the presence or the absence of errors in the program and its slices
- Formalization in Coq (10,000 LOC)
- Certified slicer in OCaml extracted from Coq

Future work:

- Consider a wider class of errors
  - expressions → uninitialized variables → ... → ACSL ?
- Extend the language
  - WHILE → unstructured control flow → ... → C ?
- Measure the benefits of relaxed slicing for verification

Demo: extracted certified slicer on an example

Slice of the program in test_prog w.r.t. line 9.

$ ./test_slice.byte test_prog \[9\]

```
Original program:
0: ASSERT (not ((x1) <= (0))) && (not ((x1) == (0))) =>> 0;
1: ASSERT (not ((x5) <= (0))) && (not ((x5) == (0))) =>> 1;
WHILE 2: not ((x5) == (0)) DO
    3: x10 := 0;
    4: x11 := x1;
    WHILE 5: (x11) <= (x5) DO
        6: x10 := (x10) + (1);
        7: x11 := (x11) + (x1)
    END;
    8: x1 := x5;
    9: x5 := x11
END
```

```
Slice:
WHILE 2: not ((x5) == (0)) DO
    4: x11 := x1;
    WHILE 5: (x11) <= (x5) DO
        7: x11 := (x11) + (x1)
    END;
    8: x1 := x5;
    9: x5 := x11
END
```