Modular, compositional and sound verification of the input/output behavior of programs

Willem Penninckx, Bart Jacobs, Frank Piessens

Department of Computer Science, KU Leuven, Belgium

DRADS 2014

Table of Contents

Introduction

Requirements

Specifying I/O behaviour

Wrapping up

Table of Contents

Introduction

Requirements

Specifying I/O behaviour

Wrapping up

Popular way to prove software properties

- Theorem:

- Possible proofs:

- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number

- Possible proofs:

- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number
  Formal:

- Possible proofs:

- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number
  Formal:

  $$\textbf{if } x > y \textbf{ then } \mathit{result} := x \textbf{ else } \mathit{result} := y$$

- Possible proofs:

- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number
  Formal:

  $$\textbf{if } x > y \textbf{ then } \mathit{result} := x \textbf{ else } \mathit{result} := y$$
  $\{\mathit{result} >= x \wedge \mathit{result} >= y\}$
- Possible proofs:
- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number
  Formal:
  $\{\}$
  $\quad$ **if** $x > y$ **then** $result := x$ **else** $result := y$
  $\{result >= x \wedge result >= y\}$
- Possible proofs:
- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number

  Formal:

  $\{\}$

  $\quad$ **if** $x > y$ **then** $result := x$ **else** $result := y$

  $\{result >= x \wedge result >= y\}$

- Possible proofs:

  No time to explain!

- When is a proof correct?

Popular way to prove software properties

- Theorem: Informal: returns a bigger number
  Formal:
  $\{\}$
  $\quad$ **if** $x > y$ **then** $result := x$ **else** $result := y$
  $\{result >= x \wedge result >= y\}$
- Possible proofs:

  No time to explain!
- When is a proof correct?
  No time to explain!

- $\underbrace{\{x = 2\}}_{\text{state before program starts}} \quad x := x + 1 \quad \underbrace{\{x = 3\}}_{\text{state after program terminates}}$
  - People added support for...
    - Concurrency
    - Dynamic memory allocation
    - ...

- $$\underbrace{\{x = 2\}}_{\text{state before program starts}} \quad x := x + 1 \quad \underbrace{\{x = 3\}}_{\text{state after program terminates}}$$

  - People added support for...
    - Concurrency
    - Dynamic memory allocation
    - ...

  - Typically verified: (memory) state.

- $\underbrace{\{x = 2\}}_{\text{state before program starts}} \quad x := x + 1 \quad \underbrace{\{x = 3\}}_{\text{state after program terminates}}$
    - People added support for...
        - Concurrency
        - Dynamic memory allocation
        - ...
    - Typically verified: (memory) state.
    - End-users care about: what's on their screen.

- $\underbrace{\{x = 2\}}_{\text{state before program starts}} \quad x := x + 1 \quad \underbrace{\{x = 3\}}_{\text{state after program terminates}}$
  - People added support for...
    - Concurrency
    - Dynamic memory allocation
    - ...
  - Typically verified: (memory) state.
  - End-users care about: what's on their screen.
- => Add support to verify Input/Output (I/O)

Table of Contents

Introduction

Requirements

Specifying I/O behaviour

Wrapping up

```
{...}
main(){
  code;
  code;
  code;
  code;
  code;
  code;
  code;
  code;
  ...
  code;
}
{...}
```

~~{...}~~

~~main(){~~

~~code;~~

~~code;~~

~~code;~~

~~code;~~

~~code;~~

~~code;~~

~~code;~~

~~code;~~

~~...~~

~~code;~~

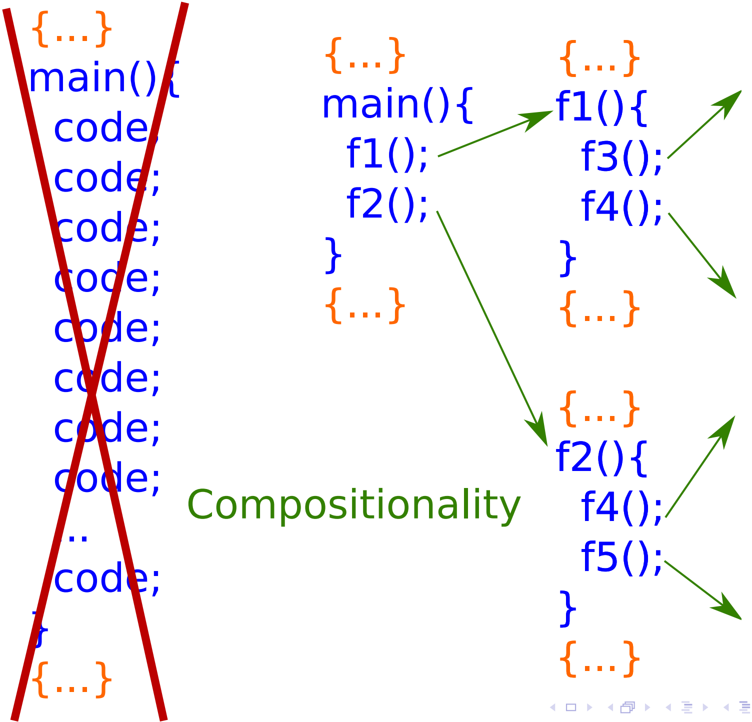
~~}~~

~~{...}~~

```
{...}
main(){
  f1();
  f2();
}
{...}
```

```
{...}
f1(){
  f3();
  f4();
}
{...}

{...}
f2(){
  f4();
  f5();
}
{...}
```



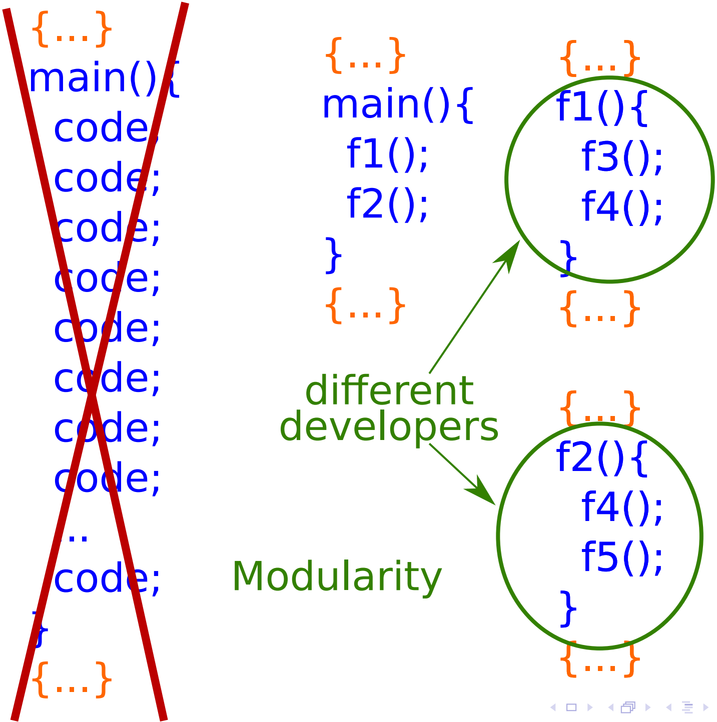
```
{...}
main(){
  code;
  code;
  code;
  code;
  code;
  code;
  code;
  code;
  ...
  code;
}
{...}
```

```
{...}
main(){
  f1();
  f2();
}
{...}
```

```
{...}
f1(){
  f3();
  f4();
}
{...}
```

```
{...}
f2(){
  f4();
  f5();
}
{...}
```

Compositionality



```
{...}
main(){
  code;
  code;
  code;
  code;
  code;
  code;
  code;
  code;
  ...
  code;
}
{...}
```

```
{...}
main(){
  f1();
  f2();
}
{...}
```

```
{...}
f1(){
  f3();
  f4();
}
{...}
```

```
{...}
f2(){
  f4();
  f5();
}
{...}
```

different developers

Modularity

```
{specs
specs
specs
specs}
main(){
...
}
{
specs
specs
specs
specs}
```

~~{specs~~

~~specs~~

~~specs~~

~~specs}~~

~~main(){~~

~~...~~

~~}~~

~~{~~

~~specs~~

~~specs~~

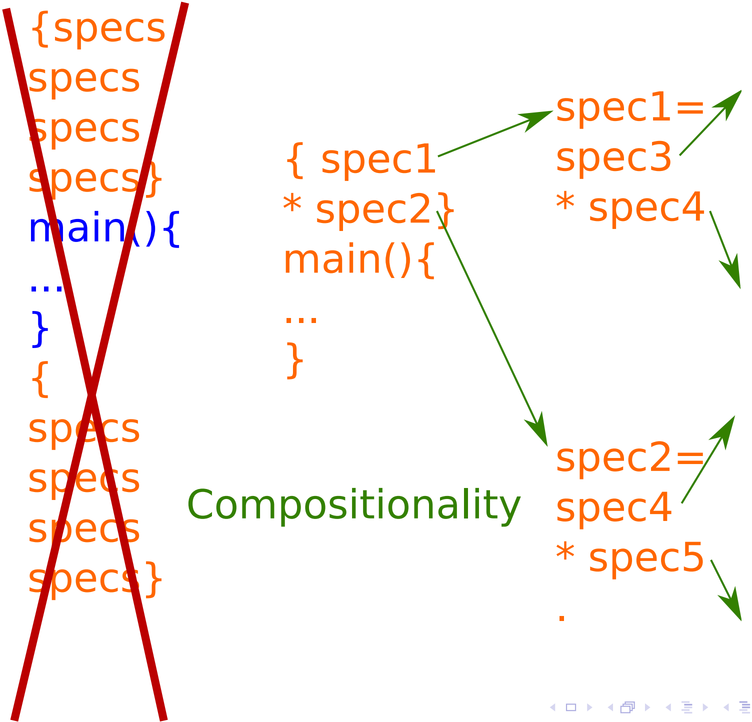
~~specs~~

~~specs}~~

```
{ spec1
* spec2}
main(){
...
}
```

```
spec1=
spec3
* spec4
```

```
spec2=
spec4
* spec5
.
```



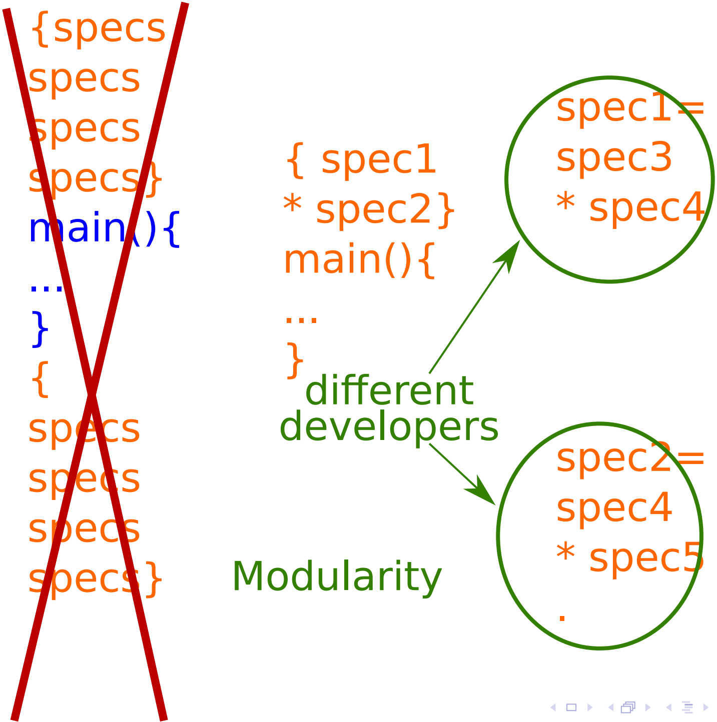
~~{specs~~
~~specs~~
~~specs~~
~~specs}~~
~~main(){~~
~~...~~
~~}~~
~~{~~
~~specs~~
~~specs~~
~~specs~~
~~specs}~~

```
{ spec1
* spec2}
main(){
...
}
```

```
spec1=
spec3
* spec4
```

```
spec2=
spec4
* spec5
.
```

Compositionality



~~{specs~~
~~specs~~
~~specs~~
~~specs}~~
~~main(){~~
~~...~~
~~}~~
~~{~~
~~specs~~
~~specs~~
~~specs~~
~~specs}~~

{ spec1
* spec2}
main(){
...
}

different developers

spec1=
spec3
* spec4

spec2=
spec4
* spec5
.

Modularity

Requirements

- Compositionality.
  - e.g. define I/O action `download` on top of `tcp_write` and `file_write`, etc.

Requirements

- Compositionality.
  - e.g. define I/O action `download` on top of `tcp_write` and `file_write`, etc.
- Modularity
  - e.g. combine independent I/O action `tcp_write` with `file_write`

Requirements

- Compositionality.
  - e.g. define I/O action `download` on top of `tcp_write` and `file_write`, etc.
- Modularity
  - e.g. combine independent I/O action `tcp_write` with `file_write`
- Non-terminating programs (part WIP)
  - e.g. {} **while true ...** {these I/O happened}: postcondition useless

Requirements

- Compositionality.
  - e.g. define I/O action `download` on top of `tcp_write` and `file_write`, etc.
- Modularity
  - e.g. combine independent I/O action `tcp_write` with `file_write`
- Non-terminating programs (part WIP)
  - e.g. {} **while true ...** {these I/O happened}: postcondition useless
- Actions depend on outcome of actions
  - e.g. read file containing filenames to read
- ...

Table of Contents

Introduction

Requirements

Specifying I/O behaviour

Wrapping up

By example

By example

- {} ... {}
  - No I/O allowed

By example

- $\{\} \dots \{\}$
  - No I/O allowed
- $\{\ \textbf{time}(t_1)\ \}\ \dots\ \{\ \textbf{time}(t_1)\ \}$
  - No I/O allowed
  - A time like $t_1 \approx$ a point in time.
  - Doing I/O “increases” time

By example

- $\{\} \ldots \{\}$
  - No I/O allowed
- $\{\ \textbf{time}(t_1)\ \}\ \ \ldots\ \ \{\ \textbf{time}(t_1)\ \}$
  - No I/O allowed
  - A time like $t_1 \approx$ a point in time.
  - Doing I/O “increases” time
- $\{\ \textbf{time}(t_1) \star \mathrm{print\_io}(t_1, \text{‘h}', t_2)\ \}$
  $\mathrm{print\_char}(\text{‘h}');$
  $\{\ \textbf{time}(t_2)\ \}$
  - Doing print_char(‘h’)
    - requires a permission $\mathrm{print\_io}(t_1, \text{‘h}', t_2)$
    - requires a $\textbf{time}(t_1)$
    - disposes the permission
    - “increases” the time to t_2

- $\{\ \mathbf{time}(t_1) \star \text{print\_io}(t_1, \text{‘h}', t_2) \star \text{print\_io}(t_2, \text{‘i}', t_3)\ \}$

  …

  $\{\ \mathbf{time}(t_3)\ \}$

  - Can print “hi”, “h”, “”.
  - If terminates: can only print “hi”.
  - Can not print: “x”, “i”, “ih”, ...

- $\{\ \textbf{time}(t_1) \star \text{print\_io}(t_1, \text{‘h'}, t_2) \star \text{print\_io}(t_2, \text{‘i'}, t_3)\ \}$

  ...

  $\{\ \textbf{time}(t_3)\ \}$
  - Can print “hi”, “h”, “”.
  - If terminates: can only print “hi”.
  - Can not print: “x”, “i”, “ih”, ...
- $\{\ \textbf{time}(t_1) \star \text{print\_io}(t_1, \text{‘h'}, t_2) \star \text{print\_io}(t_1, \text{‘i'}, t_2)\ \}$

  ...

  $\{\ \textbf{time}(t_2)\ \}$
  - Can print “h”, “i”, “”.
  - If terminates: has printed either “h” or “i”.
  - Can not print: “x”, “hi”, ...

Defining new I/O actions

```
▸ **predicate** print_string_io($t_1, str, t_2$) =
    **if** $str$ = nil **then**
        $t_1 = t_2$
    **else** (
        print_io($t_1$, head($str$), $t_{between}$)
        $\star$ print_string_io($t_{between}$, tail($str$), $t_2$)
    )
```

Defining new I/O actions

-
```
predicate print_string_io($t_1, str, t_2$) =
  if $str$ = nil then
    $t_1 = t_2$
  else (
    print_io($t_1$, head($str$), $t_{between}$)
    $\star$ print_string_io($t_{between}$, tail($str$), $t_2$)
  )
```

- Build actions using actions (compositionality)

Defining new I/O actions

-
  ```
  predicate print_string_io($t_1$, $str$, $t_2$) =
    if $str$ = nil then
      $t_1 = t_2$
    else (
      print_io($t_1$, head($str$), $t_{between}$)
      $\star$ print_string_io($t_{between}$, tail($str$), $t_2$)
    )
  ```
- Build actions using actions (compositionality)
- $\{$**time**$(t_1) \star$ print_string_io(t_1, “hello world!”, t_2)$\}$

  ...

  $\{$**time**$(t_2)\}$

Linking arguments

- $$\begin{array}{ll}\{\ \mathbf{time}(t_1) & \star\ \mathrm{read\_string\_io}(t_1, str, t_2) \\ & \star\ \mathrm{print\_string\_io}(t_2, str, t_3) \\ \} & \end{array}$$

  $\dots$

  $\{\mathbf{time}(t_3)\}$

Unconstrained order/interleaving

- $$\begin{aligned} \{\ \mathbf{time}(t_2) &\star \mathbf{time}(t_3) \\ &\star \mathrm{read\_string\_io}(t_2, str, t_4) \\ &\star \mathrm{print\_string\_io}(t_3, str, t_5) \\ \} & \\ \ldots & \\ \{\ \mathbf{time}(t_4) &\star \mathbf{time}(t_5)\ \} \end{aligned}$$
- Allows buffering of any size.

Unconstrained order/interleaving

- $\{\ \textbf{time}(t_2) \star \textbf{time}(t_3)$
  $\star\ \text{read\_string\_io}(t_2, str, t_4)$
  $\star\ \text{print\_string\_io}(t_3, str, t_5)$
  $\}$
  ...
  $\{\ \textbf{time}(t_4) \star \textbf{time}(t_5)\ \}$
- Allows buffering of any size.
- How to get two times ($\textbf{time}(t_2)$ and $\textbf{time}(t_3)$)?

Unconstrained order/interleaving

- $$\begin{aligned}\{\ &\textbf{time}(t_1) \star \textbf{split}(t_1, t_2, t_3)\\ &\qquad\star \text{read\_string\_io}(t_2, str, t_4)\\ &\qquad\star \text{print\_string\_io}(t_3, str, t_5)\\ &\qquad\star \textbf{join}(t_4, t_5, t_6)\\ \}&\\ \ldots&\\ \{\ &\textbf{time}(t_4)\ \}\end{aligned}$$
- $\textbf{split}(t_1, t_2, t_3)$ consumes $\textbf{time}(t_1)$ and yields $\textbf{time}(t_2)$ and $\textbf{time}(t_3)$.

Table of Contents

Introduction

Requirements

Specifying I/O behaviour

Wrapping up

What can we do now?

- Verify software like this:
  1. Write software
  2. Write down the wanted I/O behaviour
  3. Write a proof (add annotations)
  4. Feed to proofchecker software.
     - Output: OK or Not OK.
- With support for:
  - Modularity
  - Compositionality
  - ...

- The End