Modular Product Programs

Marco Eilers
Peter Müller
Samuel Hitz

ETH zürich

Is this function deterministic?

HYPERPROPERTY

```
def compute(xs: List[Int]) : Int {
    var res := 0;
    var i := 0
    while (i < len(xs)) {
        var x := xs[i];
        res += computeElement(x);
        i := i + 1;
    }
    return res;
}
```

```
def computeElement(x: Int) : Int
  // POST: result is deterministic
{
    ...
}
```

- Hyperproperty: Property of multiple traces of a program
- Prove: $xs_1 = xs_2 \Longrightarrow res_1 = res_2$
- Similar: Determinism, symmetry, monotonicity, transitivity

Self-composition

[Barthe et al., Math. Struc. Comp. Sci. 21(6), 2011]

```
assume xs1 == xs2;
var res1 := 0;
var i1 := 0;
while (i1 < len(xs1)) {
     var x1 := xs1[i1];
     res1 += computeElement(x1, secret1);
     i1 := i1 + 1;
}

var res2 := 0;
var i2 := 0
while (i2 < len(xs2)) {
     var x2 := xs2[i2];
     res2 += computeElement(x2, secret2);
     i2 := i2 + 1;
}
assert res1 == res2;
```

- Reduces hyperproperty to trace property
- Theoretically complete
- Non-modular

Product program

[Barthe et al., FM 2011]
[Barthe et al., LFCS 2013]

```
assume xs1 == xs2
var res1 := 0;
var res2 := 0;

var i1 := 0;
var i2 := 0;
while (i1 < len(xs1)) {
    assert i2 < len(xs2)
    var x1 := xs1[i1];
    var x2 := xs2[i2];
    res1, res2 += computeElement(x1, secret1,
                                 x2, secret2);
    i1 := i1 + 1;
    i2 := i2 + 1;
}
assert res1 == res2;
```

```
def computeElement(x1: Int, x2: Int)
    returns (res1: Int, res2: Int)
  // POST: x1 = x2 ==> res1 = res2
{
    ...
}
```

- Modular verification in special cases
  - Using relational specification
  - Requires identical control flow
- Manual construction, intertwined with specification

Product programs: limitations

```
def fac(i: Int) returns (res: Int) {
    if (i > 1) {
        rec := fac(i - 1);
        res := i * rec;
    } else {
        res := 1;
    }
}
```

- Prove monotonicity: i1 > i2 ==> res1 >= res2
- Proof requires assumption about recursive call
- Not guaranteed that both executions perform recursive call

Overview

Modular Product Programs

Goal: parameterized procedure calls

```
def fac(i: Int)
    returns (res: Int) {
    ...
    if (i > 1) {
        rec := fac(i - 1);
        ...
    }
    ...
}
```

```
def fac(i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    ...
    rec1, rec2 := fac(i1 > 0, i2 > 0,
                      i1 - 1, i2 - 1);
    ...
}
```

- Single procedure call in product
- Allows using relational specifications
- Activation variables represent which executions are active

Goal: parameterized procedure calls

```
rec1, rec2 := fac(True, True,
                  i1 - 1, i2 - 1);
```

```
rec1 := fac(i1 - 1);
rec2 := fac(i2 - 1);
```

```
rec1, rec2 := fac(True, False,
                  i1 - 1, i2 - 1);
```

```
rec1 := fac(i1 - 1);
```

```
rec1, rec2 := fac(False, True,
                  i1 - 1, i2 - 1);
```

```
rec2 := fac(i2 - 1);
```

```
rec1, rec2 := fac(False, False,
                  i1 - 1, i2 - 1);
```

```
skip;
```

Transforming the fac procedure

```
def fac(i: Int)
    returns (res: Int) {
    if (i > 1) {
        rec := fac(i - 1);
        res := i * rec;
    } else {
        res := 1;
    }
}
```

```
def fac(a1, a2, i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    ...
}
```

Transforming the fac procedure

```
def fac(i: Int)
    returns (res: Int) {
    if (i > 1) {
        rec := fac(i - 1);
        res := i * rec;
    } else {
        res := 1;
    }
}
```

```
def fac(a1, a2, i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    t1 := a1 && i1 > 1;
    t2 := a2 && i2 > 1;
    rec1, rec2 := fac(t1, t2,
                      i1 - 1, i2 - 1);
    ...
}
```

- Create new activation variables for branch
- Pass to callee

Transforming the fac procedure

```
def fac(i: Int)
    returns (res: Int) {
    if (i > 1) {
        rec := fac(i - 1);
        res := i * rec;
    } else {
        res := 1;
    }
}
```

```
def fac(a1, a2, i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    t1 := a1 && i1 > 1;
    t2 := a2 && i2 > 1;
    rec1, rec2 := fac(t1, t2,
                      i1 - 1, i2 - 1);
    if (t1) { res1 := i1 * rec1; }
    if (t2) { res2 := i2 * rec2; }
    ...
}
```

- Execute assignments conditionally

Transforming the fac procedure

```
def fac(i: Int)
    returns (res: Int) {
    if (i > 1) {
        rec := fac(i - 1);
        res := i * rec;
    } else {
        res := 1;
    }
}
```

```
def fac(a1, a2, i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    t1 := a1 && i1 > 1;
    t2 := a2 && i2 > 1;
    e1 := a1 && !(i1 > 1);
    e2 := a2 && !(i2 > 1);
    rec1, rec2 := fac(t1, t2,
                      i1 - 1, i2 - 1);
    if (t1) { res1 := i1 * rec1; }
    if (t2) { res2 := i2 * rec2; }
    if (e1) { res1 := 1; }
    if (e2) { res2 := 1; }
}
```

- Else-branch executed under different condition

Transforming the fac procedure

```
def fac(i: Int)
    returns (res: Int) {
    if (i > 1) {
        rec := fac(i - 1);
        res := i * rec;
    } else {
        res := 1;
    }
}
```

- Prevent infinite recursion

```
def fac(a1, a2, i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    t1 := a1 && i1 > 1;
    t2 := a2 && i2 > 1;
    e1 := a1 && !(i1 > 1);
    e2 := a2 && !(i2 > 1);
    if (t1 || t2) {
        rec1, rec2 := fac(t1, t2,
                          i1 - 1, i2 - 1);
    }
    if (t1) { res1 := i1 * rec1; }
    if (t2) { res2 := i2 * rec2; }
    if (e1) { res1 := 1; }
    if (e2) { res2 := 1; }
}
```

Product transformation

Statement s	[s](a1, a2)
`x := e`	`if (a1) { x1 := e1; }` `if (a2) { x2 := e2; }`

`if (c) {`
`  s1`
`} else {`
`  s2`
`}`
	`t1 := a1 && c1;`
	`t2 := a2 && c2;`
	`e1 := a1 && !c1;`
	`e2 := a2 && !c2;`
	`[s1](t1, t2)`
	`[s2](e1, e2)`

`x := m(e)`	`if (a1 || a2) {`
	`  tmp1, tmp2 := m(a1, a2, e1, e2)`
	`  if (a1) { x1 := tmp1 }`
	`  if (a2) { x2 := tmp2 }`
	`}`

Product transformation: assertions

```
def fac(a1, a2, i1: Int, i2: Int)
    returns (res1: Int, res2: Int) {
    …
    // PRE: i >= 0
    // POST-REL: i1 > i2 ==> res1 >= res2
}
```

- Unary assertions should hold for *every single* active execution

```
// PRE: a1 ==> i1 >= 0
// PRE: a2 ==> i2 >= 0
```

- Relational assertions should hold if *all* executions are active

```
// POST: a1 && a2 ==> i1 > i2 ==> res1 >= res2
```

Product transformation: loops

Statement s

```
while (e)
{
    s
}
```

[s](a1, a2)

```
while (a1 && e1 || a2 && e2)
{
    l1 := a1 && e1;
    l2 := a2 && e2;
    [s](l1, l2)
}
```

Application: Secure Information Flow

Secure Information Flow: Example

```
def compute(xs: List[Int], secret: Int)
    returns (res: Int)
    // PRE: low(xs)
    // POST: low(res)
{
    var res := 0;
    var i := 0
    while (i < len(xs))
      // INVARIANT: low(i) && low(res)
    {
        var x := xs[i];
        res += computeElement(x, secret);
        i := i + 1;
    }
    return res;
}
```

```
def computeElement(x: Int, secret: Int)
    returns (res: Int)
  // POST: low(x) ==> low(res)
{
    ...
}
```

- No functional information required
- No information about methodology

Secure Information Flow: Example

```
def compute(xs: List[Int], secret: Int)
    returns (res: Int)
    // PRE: a1 && a2 ==> xs1 == xs2
    // POST: a1 && a2 ==> res1 == res2
{
    var res := 0;
    var i := 0
    while (i < len(xs))
      // INVARIANT: low(i) && low(res)
    {
        var x := xs[i];
        res += computeElement(x, secret);
        i := i + 1;
    }
    return res;
}
```

```
def computeElement(x: Int, secret: Int)
    returns (res: Int)
  // POST: low(x) ==> low(res)
{
    ...
}
```

- No functional information required
- No information about methodology

Secure Information Flow: Events

```
def computeElement(x: Int, secret: Int) : Void
  // PRE: low(x)
{
      ...
      if (secret < 0) {
           print(0)
      }
      ...
}
```

```
def print(x: Int) : Int
  // PRE: low(x)
  // PRE: lowEvent
{
      ...
}
```

```
> computeElement(1, 45)
> computeElement(1, -45)
0
>
```

Secure Information Flow: Events

```
def computeElement(x: Int, secret: Int) : Void
  // PRE: low(x)
{
      ...
      if (secret < 0) {
           print(0)
      }
      ...
}
```

```
def print(a1, a2, x1: Int, x2: Int) : Int
  // PRE: a1 && a2 ==> x1 == x2
  // PRE: a1 == a2
{
      ...
}
```

```
> computeElement(1, 45)
> computeElement(1, -45)
0
>
```

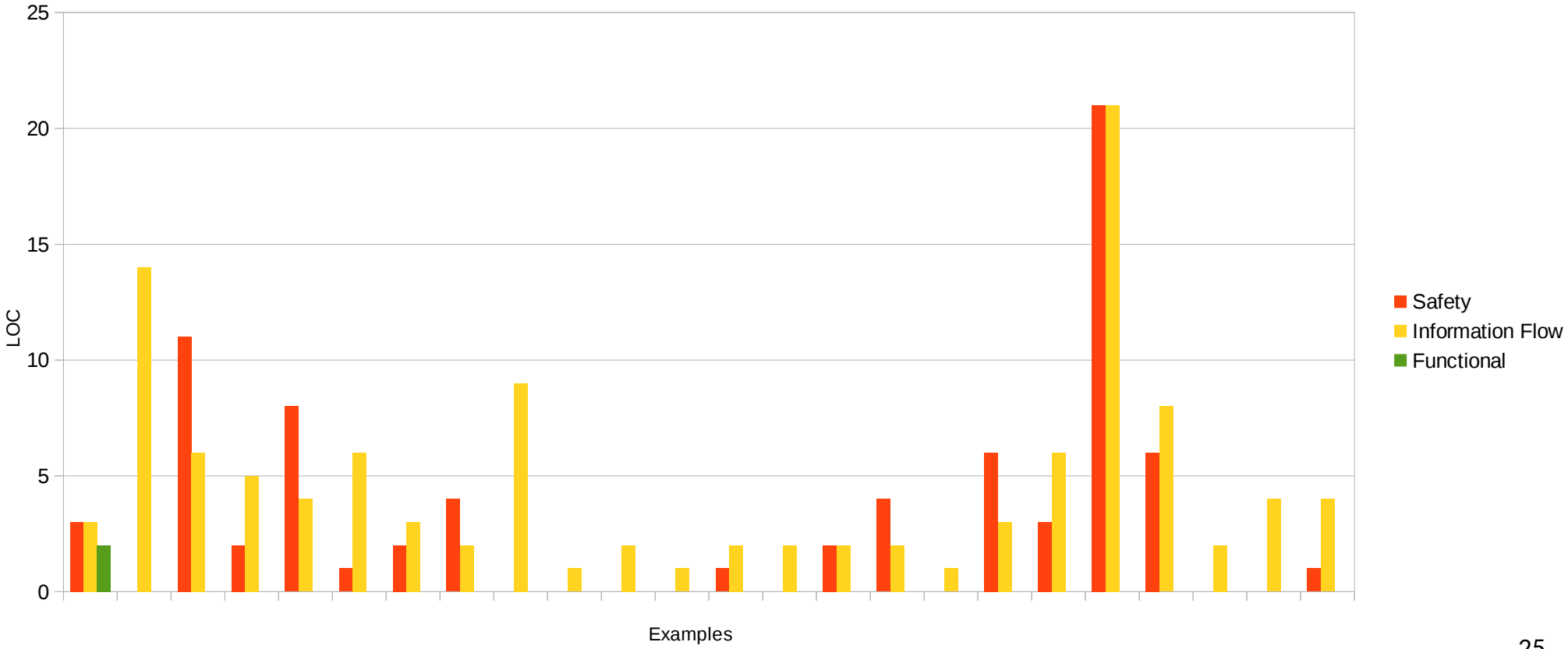
Evaluation

Implementation

```
17  method main(people: Seq[Int])
18       returns (count: Int)
19      requires low(|people|)
20      // The first bit of all entries is low
21      requires forall j: Int :: j >= 0 && j < |people| ==> low(people[j] % 2)
22      // The resulting count is low
23      ensures low(count)
24  {
25    var i: Int := 0
26    count := 0
27    while (i < |people|)
28      // Loop invariant contains only safety info (range of i)
29      invariant i >= 0 && i <= |people|
30      // and relational information about i and count, no functional spec.
31      invariant low(i)
32      invariant low(count)
33    {
34      var current: Int := people[i]
35      var female : Int
36      female := is_female(current)
37      // female is now known to be low
38      count := count + female
39      i := i + 1
40    }
41  }
```

Verification successful! Details

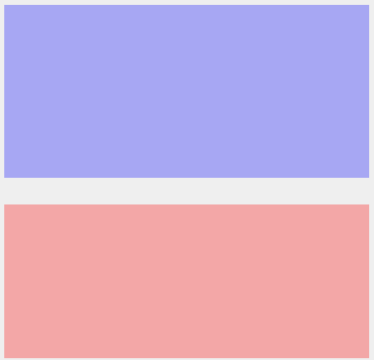
Evaluation: Required Specifications



Summary

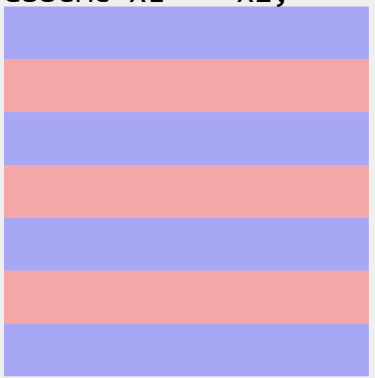
- Modular verification of arbitrary *k*-hypersafety properties
  - Usable with off-the-shelf verifiers
  - Key idea: Statement transformation adds off-switch
    - Avoid duplication of calls, loops
- Application to secure information flow
- In the paper:
  - Transformation for arbitrary *k*
  - Declassification
  - Termination/timing channels
- Future work: Inference, real languages

```
assume x1 == x2;
```



```
assert res1 == res2;
```

```
assume x1 == x2;
```



```
assert res1 == res2;
```

```
while (a1 && e1 || a2 && e2)
{
    l1 := a1 && e1;
    l2 := a2 && e2;
    [s](l1, l2)
}
```

low(e)	a1 && a2 ==> e1 = e2
lowEvent	a1 == a2