Your Proof Fails?
Testing Helps to Find the Reason

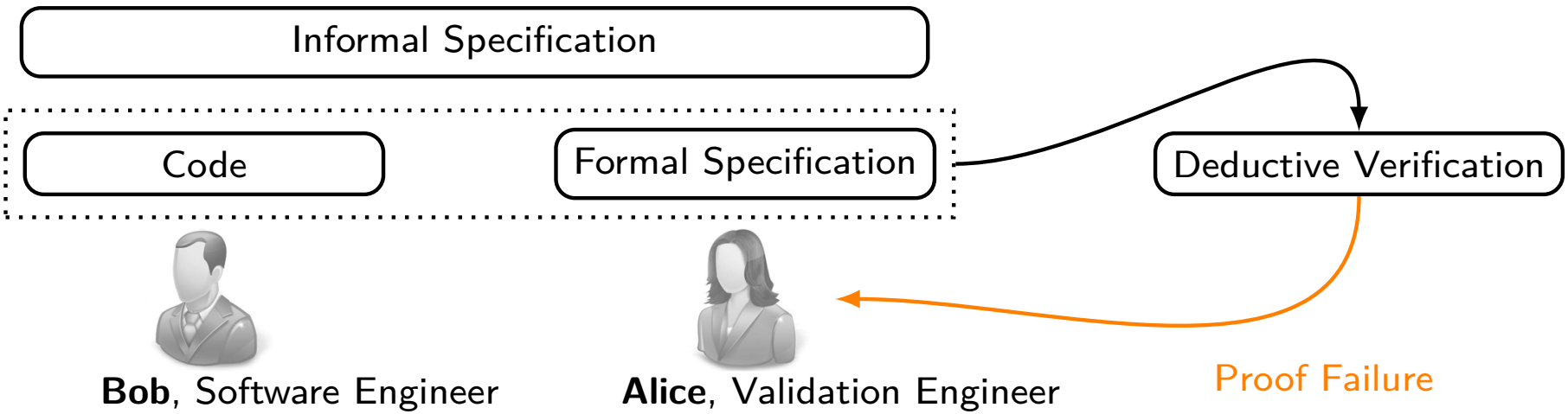
Nikolai Kosmatov[1]

joint work with Bernard Botella[1], Alain Giorgetti[2],
Jacques Julliand[2], Guillaume Petiot[1,2]

[1]CEA List
[2]FEMTO-ST, Univ. of Franche-Comté

TAP 2016, Vienna, July 6, 2016

Global Motivation: Facilitate Software Verification

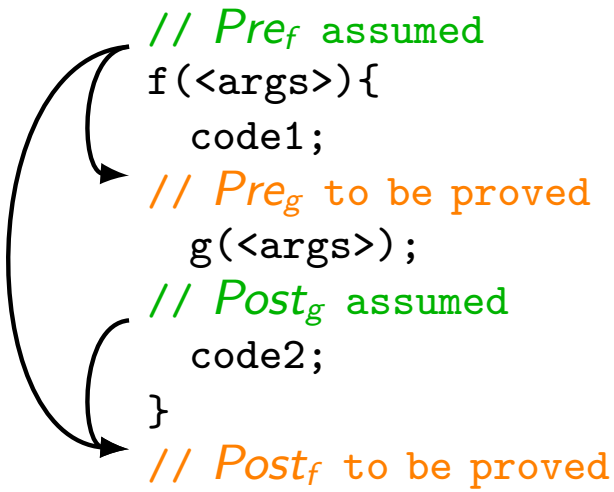


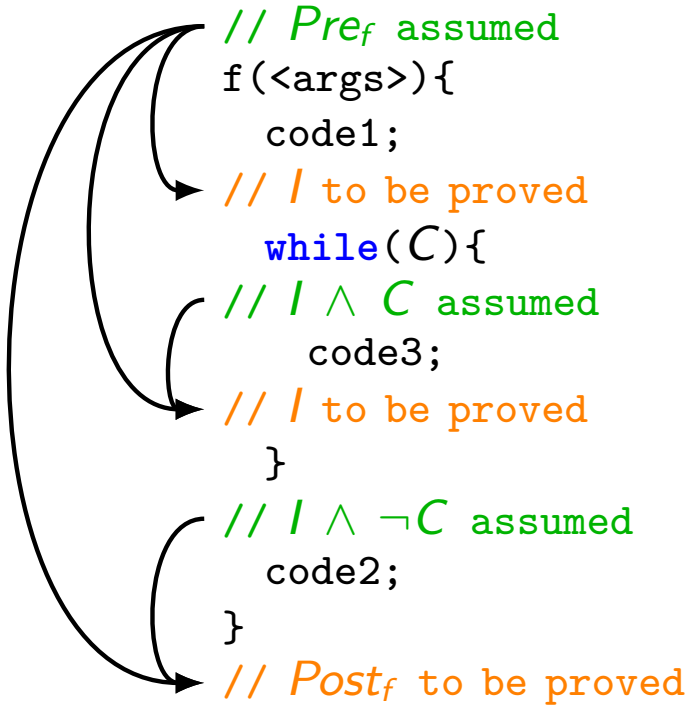
Why does my proof fail?

Analysis of proof failures is costly and often requires

- deep knowledge of provers
- careful review of code / specification
- interactive proof in a proof assistant

Modular Deductive Verification in a Nutshell





A proof failure can be due to various reasons!

For convenience, we say:
A *subcontract* of f is the contract of a called function or loop in f.

Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Can be proven with Frama-C/WP

Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Postcondition unproven...

...because a loop invariant is missing.

Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result == 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 1;
}
```

Postcondition unproven...

...because it is incorrect.

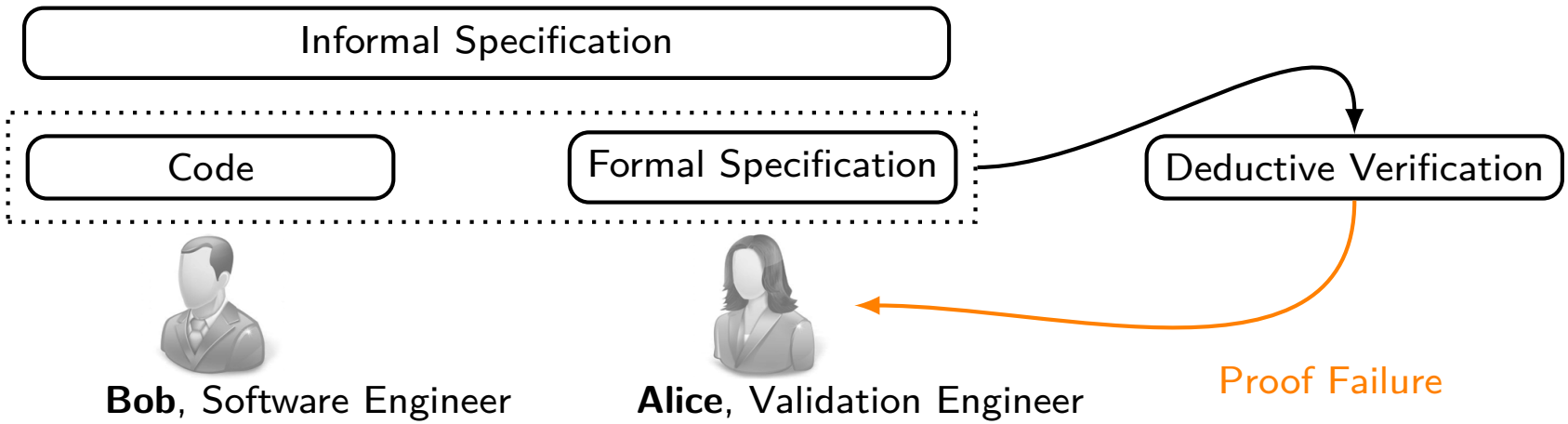
Example: Several reasons for the same proof failure

```
/*@ requires n>=0 && \valid(t+(0..n-1));
    assigns \nothing;
    ensures \result != 0 <==>
      (\forall integer j; 0 <= j < n ==> t[j] == 0);
*/
int all_zeros(int t[], int n) {
  int k;
  /*@ loop invariant 0 <= k <= n;
      loop invariant \forall integer j; 0<=j<k ==> t[j]==0;
      loop assigns k;
      loop variant n-k;
  */
  for(k = 0; k < n; k++)
    if (t[k] != 0)
      return 0;
  return 0;
}
```

Postcondition unproven...

...because the code is incorrect.

What is the right way to go?



Code non-compliant to spec?
- error in the code?
- error in the spec?

Subcontract weakness?
- too weak loop invariant?
- too weak contract of a callee?

Prover incapacity?
- add assertions, lemmas...?
- use interactive proof?

Our main goals: a complete verification methodology to

- automatically and precisely diagnose proof failures,
- provide a counter-example to illustrate the issue

Outline

Plan

(Selected) Related Work

Solver-based counter-examples:

- model-checkers [Torlak, TACAS'07], CBMC [Groce, CAV'14]
- proof assistant Isabelle/NitPick [Blanchette, ITP'10]

In verification tools:

Modular vision: solver-based counter-examples

- ESC/Java [Leino, SCP'05], OpenJML [Cok, TCS'14],
- Dafny/Boogie [Le Goues, SEFM'11], [Leino, F-IDE'14]
- SPARK [Hauzar, SEFM'16]

Non-modular vision: code-based counter-examples

- by testing: Frama-C/STADY [Petiot, TAP'14, SCAM'14]
- by testing: Dafny/Delfy [Christakis, TACAS'16]
- by inlining/unrolling: AutoProof [Tschannen, VSTT'14]

Proof tree analysis in KeY [Engel, TAP'07], [Gladisch, TAP'09]

This work: a complete testing based approach from both modular and non-modular perspective to diagnose all kinds of failures

Plan

Frama-C at a glance

- A platform for analysis of C code
- Developed at CEA List in collaboration with INRIA Saclay
- ACSL annotation language
- Extensible plugin oriented platform
  - Collaboration of analyses over same code
  - Inter plugin communication through ACSL formulas
  - Adding specialized plugins is easy
- Used in industry (avionics, energy, rail,...)
- `http://frama-c.com/` [Kirchner et al. FAC 2015]

ACSL Specification Language

ACSL: **A**NSI/ISO **C** **S**pecification **L**anguage

- Based on the notion of contract, like in Eiffel, JML
- Expresses functional properties
  - first-order logic
  - Pure C expressions and ACSL terms
  - C types + $\mathbb{Z}$ (`integer`) + $\mathbb{R}$ (`real`)
- `http://frama-c.com/acsl`

E-ACSL: executable subset of ACSL

- verifiable in finite time
- only finite quantifications, no axioms, lemmas,...

Plugin WP for deductive verification

- Based on Weakest Precondition calculus [Dijkstra, 1976]
- Proves that a given program respects its specification

Plugin PathCrawler for test generation

- Performs Dynamic Symbolic Execution (DSE)
- Automatically creates test data to cover program paths (explored in depth-first search)
  - see [Williams, ASE'04, EDCC'05], [Botella et al. AST 2009]
- Similar to PEX, DART/CUTE, KLEE, SAGE, etc.
- Exact semantics: doesn't approximate path constraints
- Online version: `pathcrawler-online.com`

Plan

Three Kinds of Proof Failures

Non-Compliance

- a direct conflict between code and spec, confirmed by runtime assertion checking on a test input (counter-ex.)
- non-modular, code-based vision of all callees and loops

Subcontract Weakness

- occurs when a test input is a counter-ex. in the modular vision (for some callees / loops) without being a non-compliance counter-ex.

Prover Incapacity

- a proof failure when there exist neither Non-Compliance counter-ex. nor Subcontract Weakness counter-ex.

Plan

Overview of the Method

Main idea:

- instrument the program (by a spec-to-code translation), and
- use testing (DSE) to produce a counter-example

Steps:

1. Non-Compliance Detection ($\mathfrak{D}^{\mathrm{NC}}$)
2. Subcontract Weakness Detection ($\mathfrak{D}^{\mathrm{SW}}$)
   - If a counter-ex. found, check if it is a Non-Compliance counter-ex. using runtime assertion checking
3. If neither Non-Compliance nor Subcontract Weakness counter-ex. exist, this is a Prover Incapacity

Instrumentation for Non-Compliance Detection: A Function Contract

```
/*@ requires Pre_g;
    ensures Post_g; */
Type_f g(...) {
  code1;
}
```

→

```
Type_f g(...) {
  fassert(Pre_g);
  code1;
  fassert(Post_g);
}
```

Principle:

- translate annotations into C code, similarly to runtime assertion checking, but in a way that DSE can trigger errors
- details in [Petiot, SCAM'14]

Instrumentation for Subcontract Weakness Detection: A Function Call "Replaced" by its Contract

```
/*@ assigns x1,..,xN;
    ensures Post_g; */

Type_g g(...) {
  code3;
}

Type_f f(...) {
  code1;
  g(Args);
  code2;
}
```

→

```
Type_g g_sw(...) {
  x1 = NonDet();
  ...
  xN = NonDet();
  Type_g ret=NonDet();
  fassume(Post_g);
  return ret;
} //respects contract of g
Type_g f(...) {
  code1
  g_sw(Args);
  code2;
}
```

Principle:

- Replace the callee/loop code by the most general code respecting its contract, then try to trigger errors with DSE
- requires (`loop`) `assigns` clauses

Plan

Focus: Subcontract Weaknesses

- Weak subcontracts can be OK if the proof succeeds
- How to treat them if they are too weak: all or one at a time?

- We need to identify which individual subcontract is too weak
- Looking for single subcontract weaknesses is necessary!
- Is it sufficient ? No, we need to look for both single and global ones!

Global vs. single subcontract weaknesses

```
int x;

/* @ ensures x >= \old(x) + 1; assigns x; */
void g1 () { x = x + 2; }

/* @ ensures x >= \old(x) + 1; assigns x; */
void g2 () { x = x + 2; }

/* @ ensures x >= \old(x) + 1; assigns x; */
void g3 () { x = x + 2; }

/* @ ensures x >= \old(x) + 4; assigns x; */
void f () { g1(); g2(); g3(); }
```

- No single subcontract is too weak, whereas the three subcontracts together are too weak
- Detected only if all calls/loops are replaced by subcontracts
- Absence of single SWs does not imply absence of global SWs

Plan

Implementation and Experiments

- StaDy tool (on top of WP and PathCrawler in Frama-C)
- experiments on 20 annotated programs [Burghardt, et al.]
- 928 mutants generated (wrong code, wrong or missing spec)
- StaDy applied to classify failures in 848 unproven mutants

Classification:

- StaDy classified 97.2% of proof failures

Execution time comparable to WP

- Wp: in avg 2.6 s. per mutant (13 s. per unproven mutant)
- StaDy: in average 2.7 s. per unproven mutant

Partial coverage:

- Classification remains efficient even with partial coverage

Plan

Summary and Future Work

Summary:

- Three kinds of proof failures: non-compliance (NC), subcontract weakness (SW), prover incapacity
- Testing (DSE) efficiently helps to explain proof failures
- Both modular and non-modular vision of the program
- Support of global vs. single subcontract weaknesses for both generality and precision

Future Work:

- Optimized combinations of $\mathfrak{D}^{\mathrm{NC}}$ and $\mathfrak{D}^{\mathrm{SW}}$ (heuristics?)
- Extend StaDy to yet unsupported features of ACSL
- Compare to/combine with other approaches (solver-based counter-examples, inlining/unrolling...)
- Further evaluation (bigger examples, user studies...)