# Synthesis of Circular Compositional Program Proofs via Abduction

Boyang Li, Işıl Dillig, Tom Dillig (College of William & Mary)
Ken McMillan ( Microsoft Research)
Mooly Sagiv (Tel Aviv University)

# Motivation

- Different verification approaches have various strengths and weaknesses

# Motivation

- Different verification approaches have various strengths and weaknesses
- Examples:

# Motivation

- Different verification approaches have various strengths and weaknesses
- Examples:
  - Polyhedra domain is good at inferring linear invariants

# Motivation

- Different verification approaches have various strengths and weaknesses
- Examples:
  - Polyhedra domain is good at inferring linear invariants
  - CEGAR based model checking good at separating paths in programs

# Motivation

- Different verification approaches have various strengths and weaknesses
- Examples:
  - Polyhedra domain is good at inferring linear invariants
  - CEGAR based model checking good at separating paths in programs
  - Interval analysis scales to very large programs

# Motivation

- Different verification approaches have various strengths and weaknesses
- Examples:
  - Polyhedra domain is good at inferring linear invariants
  - CEGAR based model checking good at separating paths in programs
  - Interval analysis scales to very large programs
  - ...

# Motivation

- Different verification approaches have various strengths and weaknesses
- Examples:
  - Polyhedra domain is good at inferring linear invariants
  - CEGAR based model checking good at separating paths in programs
  - Interval analysis scales to very large programs
  - ...

**Difficult, if not impossible, to design one approach that is good at everything**

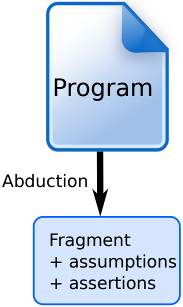
# This Talk

**New technique for circular compositional program verification**

# This Talk

**New technique for circular compositional program verification**

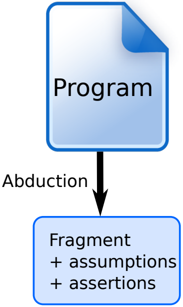
- Decompose the program proofs into small lemmas using logical abduction



# This Talk

**New technique for circular compositional program verification**

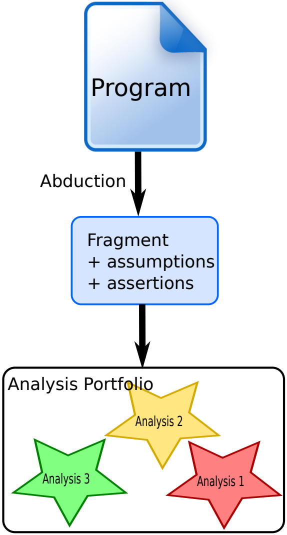
- Decompose the program proofs into small lemmas using logical abduction
- Represent lemmas as code fragments annotated with assertions and assumptions



# This Talk

**New technique for circular compositional program verification**

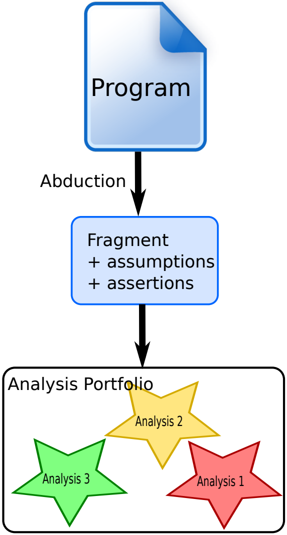
- Decompose the program proofs into small lemmas using logical abduction
- Represent lemmas as code fragments annotated with assertions and assumptions
- Use portfolio of verification techniques to discharge fragments



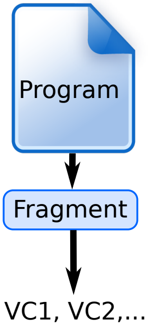
# This Talk

**New technique for circular compositional program verification**

- Decompose the program proofs into small lemmas using logical abduction
- Represent lemmas as code fragments annotated with assertions and assumptions
- Use portfolio of verification techniques to discharge fragments
- Use circular compositional reasoning to turn some assertions into assumptions

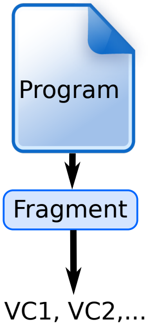


# Proof Decomposition



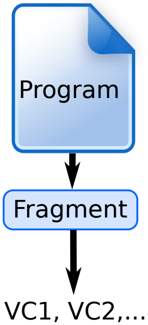

- Compute VCs of assertion on program fragment

# Proof Decomposition



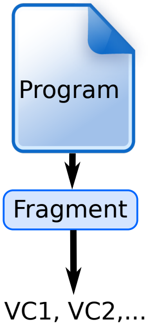

- Compute VCs of assertion on program fragment
- For any VC of the form $\phi_1 \Rightarrow \phi_2$ that is not valid, find $\psi$ such that $(\psi \wedge \phi_1) \Rightarrow \phi_2$ is valid using abduction.

# Proof Decomposition




- Compute VCs of assertion on program fragment
- For any VC of the form $\phi_1 \Rightarrow \phi_2$ that is not valid, find $\psi$ such that $(\psi \wedge \phi_1) \Rightarrow \phi_2$ is valid using abduction.
- Now, introduce $\psi$ as new assertion in program

# Proof Decomposition




- Compute VCs of assertion on program fragment
- For any VC of the form $\phi_1 \Rightarrow \phi_2$ that is not valid, find $\psi$ such that $(\psi \wedge \phi_1) \Rightarrow \phi_2$ is valid using abduction.
- Now, introduce $\psi$ as new assertion in program
- And eliminate old assertion by proving it assuming $\psi$ and converting it to an assumption

# Example

- Consider the following code snippet

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

# Example

- Consider the following code snippet
- Code contains assertion in second loop

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

# Example

- Consider the following code snippet
- Code contains assertion in second loop
- Goal: Discharge assertion using portfolio of analyses on fragments of this code

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

# Decomposition

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Want to verify assertion only using highlighted fragment

# Decomposition

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Want to verify assertion only using highlighted fragment
- But not possible since precondition "$z$ is odd" is missing

# Decomposition

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
while(*) {
  assert(x==y);

  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Want to verify assertion only using highlighted fragment
- But not possible since precondition “$z$ is odd” is missing

**Want to solve for missing assumptions required to prove $x = y$**

# Parametric VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Use $\phi_1$ and $\phi_2$ to represent unknown assumptions that make the assertion valid

# Parametric VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Use $\phi_1$ and $\phi_2$ to represent unknown assumptions that make the assertion valid
- Compute VCs of $x = y$ parametric on $\phi_1$ and $\phi_2$

# Parametric VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Use $\phi_1$ and $\phi_2$ to represent unknown assumptions that make the assertion valid
- Compute VCs of $x = y$ parametric on $\phi_1$ and $\phi_2$
- VC 1:

$$(z = i - j \land x = 0 \land y = 0 \land w = 0 \land \phi_1) \Rightarrow x = y$$

# Parametric VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

$\sigma$

- Use $\phi_1$ and $\phi_2$ to represent unknown assumptions that make the assertion valid
- Compute VCs of $x = y$ parametric on $\phi_1$ and $\phi_2$
- VC 1:

$$(z = i - j \wedge x = 0 \wedge y = 0 \\ \wedge w = 0 \wedge \phi_1) \Rightarrow x = y$$

- VC 2:

$$(\phi_2 \wedge x = y) \Rightarrow wp(\sigma, x = y)$$

# Parametric VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

$\sigma$

- Use $\phi_1$ and $\phi_2$ to represent unknown assumptions that make the assertion valid
- Compute VCs of $x = y$ parametric on $\phi_1$ and $\phi_2$
- VC 1: VALID

$$(z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge \phi_1) \Rightarrow x = y$$

- VC 2:

$$(\phi_2 \wedge x = y) \Rightarrow wp(\sigma, x = y)$$

# Parametric VC Generation

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
  assert(x==y);
  assume(φ2);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

$\sigma$

- Use $\phi_1$ and $\phi_2$ to represent unknown assumptions that make the assertion valid
- Compute VCs of $x = y$ parametric on $\phi_1$ and $\phi_2$
- VC 1: VALID

$$(z = i - j \wedge x = 0 \wedge y = 0 \wedge w = 0 \wedge \phi_1) \Rightarrow x = y$$

- VC 2: NOT VALID

$$(\phi_2 \wedge x = y) \Rightarrow wp(\sigma, x = y)$$

# Finding Auxiliary Lemmas

- First, use definition of *wp* to expand VC 2

# Finding Auxiliary Lemmas

- First, use definition of *wp* to expand VC 2

$$(\phi_2 \land x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

# Finding Auxiliary Lemmas

- First, use definition of *wp* to expand VC 2

$$(\phi_2 \wedge x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

- To prove VC 2, we need to find a $\phi_2$ that makes it valid

# Finding Auxiliary Lemmas

- First, use definition of *wp* to expand VC 2

$$(\phi_2 \wedge x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

- To prove VC 2, we need to find a $\phi_2$ that makes it valid
- But $\phi_2$ should not contradict $x = y$ (lemma we want to prove)

# Finding Auxiliary Lemmas

- First, use definition of *wp* to expand VC 2

$$(\phi_2 \wedge x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

- To prove VC 2, we need to find a $\phi_2$ that makes it valid
- But $\phi_2$ should not contradict $x = y$ (lemma we want to prove)
- Therefore, want $\phi_2 \wedge x = y$ to be satisfiable

# Finding Auxiliary Lemmas

- First, use definition of *wp* to expand VC 2

$$(\phi_2 \land x = y) \Rightarrow x + (z + x + y + w)\%2 = y + 1$$

- To prove VC 2, we need to find a $\phi_2$ that makes it valid
- But $\phi_2$ should not contradict $x = y$ (lemma we want to prove)
- Therefore, want $\phi_2 \land x = y$ to be satisfiable

**Insight: This is an instance of logical abduction**

# Abductive Inference


abduce

- Given known facts $F$ and desired outcome $O$, abductive inference finds simple explanatory hypothesis $E$ such that

$$F \wedge E \models O \quad \text{and} \quad \mathrm{SAT}(F \wedge E)$$

# Abductive Inference

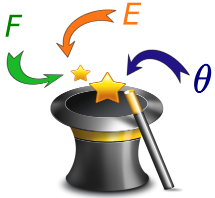

abduce

- Given known facts $F$ and desired outcome $O$, abductive inference finds simple explanatory hypothesis $E$ such that

$$F \wedge E \models O \quad \text{and} \quad \mathrm{SAT}(F \wedge E)$$

- Use abduction to generate simple assumptions that make verification condition valid

# Abductive Inference



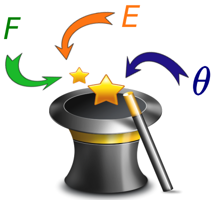

abduce

- Given known facts $F$ and desired outcome $O$, abductive inference finds simple explanatory hypothesis $E$ such that

$$F \land E \models O \quad \text{and} \quad \mathrm{SAT}(F \land E)$$

- Use abduction to generate simple assumptions that make verification condition valid
- Known facts $F$ is verification condition, desired outcome is $true$

# Abductive Inference




abduce

- Given known facts $F$ and desired outcome $O$, abductive inference finds simple explanatory hypothesis $E$ such that

$$F \wedge E \models O \quad \text{and} \quad \mathrm{SAT}(F \wedge E)$$

- Use abduction to generate simple assumptions that make verification condition valid
- Known facts $F$ is verification condition, desired outcome is $true$
- Abductive solution becomes lemma in proof and can now be established separately

# Abductive Inference in Example

- Here, for

  $$(\phi_2 \wedge x = y) \Rightarrow$$
  $$x + (z + x + y + w)\%2 = y + 1$$

  we compute the solution
  $\phi_2 : (w + z)\%2 = 1$

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assume(φ1);
while(*) {
   assert(x==y);
   assume(φ2);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

$\sigma$

# Abductive Inference in Example

- Here, for

  $$(\phi_2 \wedge x = y) \Rightarrow$$
  $$x + (z + x + y + w)\%2 = y + 1$$

  we compute the solution
  $\phi_2 : (w + z)\%2 = 1$

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assert(x==y);
   assume((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

# Abductive Inference in Example

- Here, for

  $$(\phi_2 \wedge x = y) \Rightarrow$$
  $$x + (z + x + y + w)\%2 = y + 1$$

  we compute the solution
  $\phi_2 : (w + z)\%2 = 1$

- Can now show $x = y$, which turns into an assumption

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assert(x==y);
   assume((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

# Abductive Inference in Example

- Here, for

  $$(\phi_2 \wedge x = y) \Rightarrow$$
  $$x + (z + x + y + w)\%2 = y + 1$$

  we compute the solution
  $\phi_2 : (w + z)\%2 = 1$

- Can now show $x = y$, which turns into an assumption

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assume(x==y);
   assume((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

# Abductive Inference in Example

- Here, for

  $$(\phi_2 \wedge x = y) \Rightarrow$$
  $$x + (z + x + y + w)\%2 = y + 1$$

  we compute the solution
  $\phi_2 : (w + z)\%2 = 1$

- Can now show $x = y$, which turns into an assumption

- But still need to prove $\phi_2 \Rightarrow$ add as assertion

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

# Abductive Inference in Example

- Here, for

  $$(\phi_2 \wedge x = y) \Rightarrow$$
  $$x + (z + x + y + w)\%2 = y + 1$$

  we compute the solution
  $\phi_2 : (w + z)\%2 = 1$

- Can now show $x = y$, which turns into an assumption

- But still need to prove $\phi_2 \Rightarrow$ add as assertion

- Circular compositional reasoning at work!

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

# Next Step

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

- New assertion still not provable since value of $z$ unconstrained

# Next Step

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;

while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

- New assertion still not provable since value of $z$ unconstrained
- Again generate parametric VCs

# Next Step

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(φ1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- New assertion still not provable since value of $z$ unconstrained
- Again generate parametric VCs
- First VC introduces assertion before loop

# Next Step

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(φ1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- New assertion still not provable since value of $z$ unconstrained
- Again generate parametric VCs
- First VC introduces assertion before loop

$$(\phi_1 \wedge z - i - j \wedge x = 0 \\ \wedge y = 0 \wedge w = 0 \wedge x = y) \Rightarrow \\ (w + z\%2 = 1)$$

# Next Step

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(φ1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- New assertion still not provable since value of $z$ unconstrained
- Again generate parametric VCs
- First VC introduces assertion before loop

  $$(\phi_1 \wedge z - i - j \wedge x = 0 \\ \wedge y = 0 \wedge w = 0 \wedge x = y) \Rightarrow \\ (w + z\%2 = 1)$$

- Solution computed via abduction $\phi_1 : z\%2 = 1$

# Invoking Client Analyses

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Now left with two assertions in the code fragment

# Invoking Client Analyses

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

- Now left with two assertions in the code fragment
- Convert first assertion to assumption and invoke our client analyses

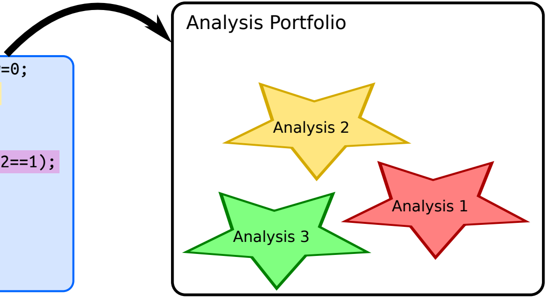
# Invoking Client Analyses

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
while(*) {
   assume(x==y);
   assert((w+z)%2==1);
   z+=x+y+w;
   y++;
   x+=z%2;
   w+=2;
}
```

- Now left with two assertions in the code fragment
- Convert first assertion to assumption and invoke our client analyses
- Again, circular compositional reasoning at work

# Invoking Client Analyses

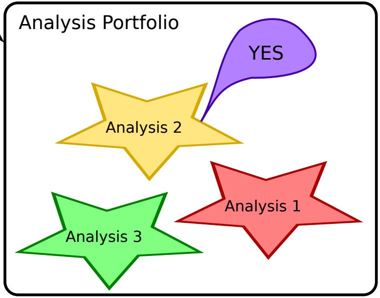
```
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```




- Give fragment with assumptions and assertions to clients

# Invoking Client Analyses

```
int x=0, y=0, w=0;
assume(z%2==1);
while(*) {
  assume(x==y);
  assert((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```




- Give fragment with assumptions and assertions to clients
- Fragment can be locally verified by divisibility analysis

## Remaining Code

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
while(*) {
  assume(x==y);
  assume((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Now, only one assertion left in our program

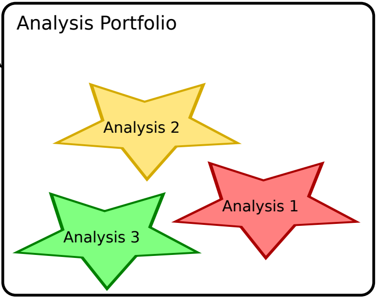
# Remaining Code

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
while(*) {
  assume(x==y);
  assume((w+z)%2==1);
  z+=x+y+w;
  y++;
  x+=z%2;
  w+=2;
}
```

- Now, only one assertion left in our program
- Extract next fragment for this assertion and give to client analyses

# Invoking Client Analyses

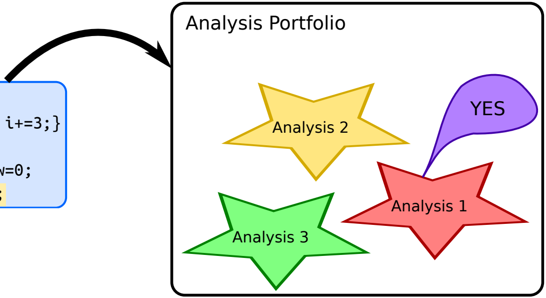
```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
```




- Invoke clients on current fragment

# Invoking Client Analyses

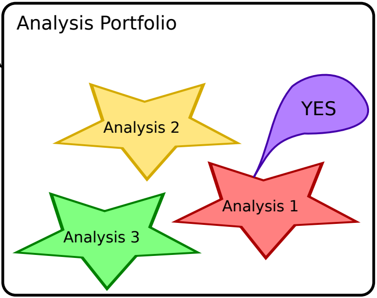
```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
```




- Invoke clients on current fragment
- This assertion can be shown by any client analysis that can establish $i = 3j + 1$

# Invoking Client Analyses

```
int i=1, j=0;
while(*) {j++; i+=3;}
int z = i-j;
int x=0, y=0, w=0;
assert(z%2==1);
```




- Invoke clients on current fragment
- This assertion can be shown by any client analysis that can establish $i = 3j + 1$

**We have now proven the original assertion**

# Our Technique at a High Level

- Technique decomposes proof of program into subgoals on syntactic fragments

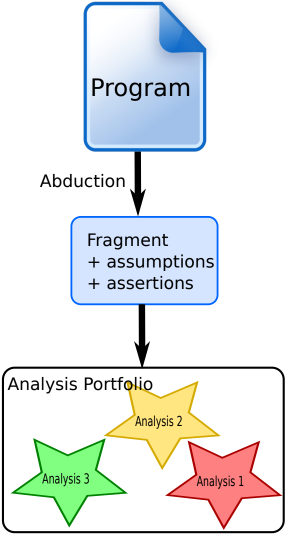
# Our Technique at a High Level

- Technique decomposes proof of program into subgoals on syntactic fragments
- While we show one subgoal, we can safely assume the others ⇒ circular reasoning

# Our Technique at a High Level

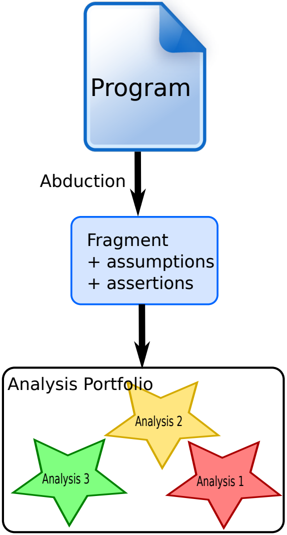
- Technique decomposes proof of program into subgoals on syntactic fragments
- While we show one subgoal, we can safely assume the others $\Rightarrow$ circular reasoning
- If a subgoal cannot be shown by any client analysis, we backtrack and generate new subgoals using abductive inference

# Advantages of this Approach



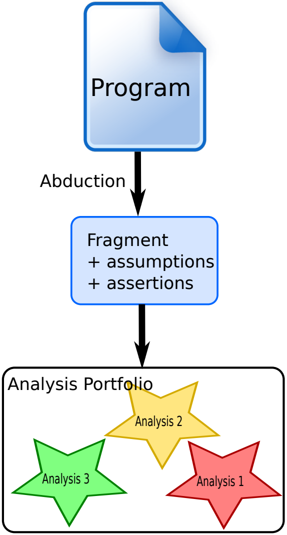

- Clients only analyze (typically) small fragments

# Advantages of this Approach



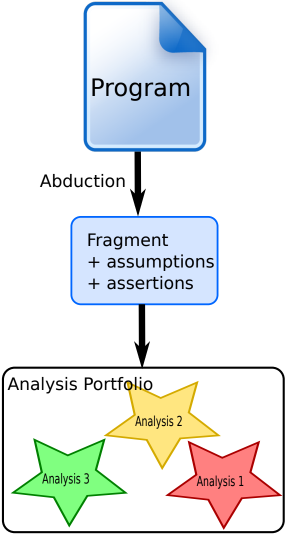

- Clients only analyze (typically) small fragments
- Interaction is demand-driven (i.e., lazy)

# Advantages of this Approach



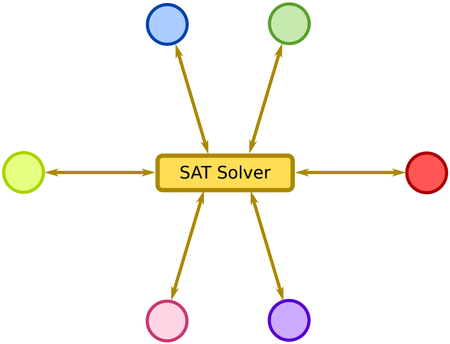

- Clients only analyze (typically) small fragments
- Interaction is demand-driven (i.e., lazy)
- Analyses with complementary strengths can help each other

# Advantages of this Approach



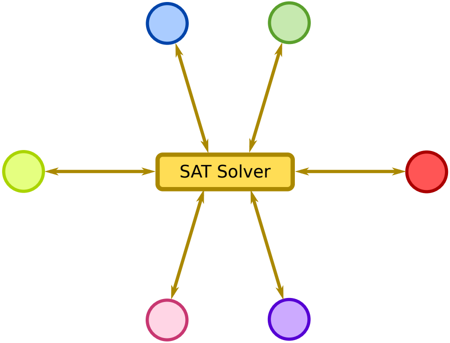

- Clients only analyze (typically) small fragments
- Interaction is demand-driven (i.e., lazy)
- Analyses with complementary strengths can help each other
- Can prove properties no client analysis or eager combination can prove alone

# Analogy to SMT Solver




- Can view client analyses as theory solvers

# Analogy to SMT Solver




- Can view client analyses as theory solvers
- Proving assertion on program invokes clients on fragments and speculated subgoals, backtracks when needed

# Experiments

- Implemented this techique and used four client tools:

# Experiments

- Implemented this techique and used four client tools:
    - Interproc Polyhedra

## Experiments

- Implemented this techique and used four client tools:
  - Interproc Polyhedra
  - Interproc Linear Congruence

# Experiments

- Implemented this techique and used four client tools:
  - Interproc Polyhedra
  - Interproc Linear Congruence
  - Blast

# Experiments

- Implemented this techique and used four client tools:
  - Interproc Polyhedra
  - Interproc Linear Congruence
  - Blast
  - Compass

# Experiments

- Implemented this techique and used four client tools:
  - Interproc Polyhedra
  - Interproc Linear Congruence
  - Blast
  - Compass

**These tools have very different strengths and weaknesses**

# First Experiment

- 10 challenging micro-benchmarks with one assertion each

# First Experiment

- 10 challenging micro-benchmarks with one assertion each
- No tool can individually prove any benchmark

# First Experiment

- 10 challenging micro-benchmarks with one assertion each
- No tool can individually prove any benchmark

| Name | LOC | Time (s) | # queries | Polyhedra | Linear Cong | Blast | Compass |
|---|---|---|---|---|---|---|---|
| B1 | 45 | 0.6 | 2 | ✘ | ✘ | ✔ | ✘ |
| B2 | 37 | 0.2 | 2 | ✘ | ✔ | ✘ | ✘ |
| B3 | 51 | 1.0 | 2 | ✔ | ✘ | ✔ | ✘ |
| B4 | 59 | 0.4 | 3 | ✔ | ✘ | ✔ | ✘ |
| B5 | 89 | 0.6 | 3 | ✔ | ✘ | ✔ | ✘ |
| B6 | 60 | 0.5 | 5 | ✘ | ✔ | ✘ | ✔ |
| B7 | 56 | 0.6 | 2 | ✘ | ✘ | ✔ | ✔ |
| B8 | 45 | 0.2 | 2 | ✔ | ✘ | ✔ | ✘ |
| B9 | 59 | 0.5 | 1 | ✘ | ✘ | ✔ | ✘ |
| B10 | 47 | 0.2 | 2 | ✔ | ✘ | ✔ | ✔ |

# First Experiment

- 10 challenging micro-benchmarks with one assertion each
- No tool can individually prove any benchmark

| Name | LOC | Time (s) | # queries | Polyhedra | Linear Cong | Blast | Compass |
|---|---|---|---|---|---|---|---|
| B1 | 45 | 0.6 | 2 | ✘ | ✘ | ✔ | ✘ |
| B2 | 37 | 0.2 | 2 | ✘ | ✔ | ✘ | ✘ |
| B3 | 51 | 1.0 | 2 | ✔ | ✘ | ✔ | ✘ |
| B4 | 59 | 0.4 | 3 | ✔ | ✘ | ✔ | ✘ |
| B5 | 89 | 0.6 | 3 | ✔ | ✘ | ✔ | ✘ |
| B6 | 60 | 0.5 | 5 | ✘ | ✔ | ✘ | ✔ |
| B7 | 56 | 0.6 | 2 | ✘ | ✘ | ✔ | ✔ |
| B8 | 45 | 0.2 | 2 | ✔ | ✘ | ✔ | ✘ |
| B9 | 59 | 0.5 | 1 | ✘ | ✘ | ✔ | ✘ |
| B10 | 47 | 0.2 | 2 | ✔ | ✘ | ✔ | ✔ |

**But all benchmarks can be proven when analyses are combined using our technique**

## Second Experiment

We also verified a complicated assertion each on five real programs

# Second Experiment

We also verified a complicated assertion each on five real programs

| Name | LOC | Time (s) | # queries | Avg # vars in query | Avg LOC in query |
|---|---|---|---|---|---|
| Wizardpen Linux Driver | 1242 | 3.8 | 5 | 1.5 | 29 |
| OpenSSH clientloop | 1987 | 2.8 | 3 | 2.3 | 5 |
| Coreutils su | 1057 | 3.0 | 5 | 1.7 | 6 |
| GSL Histogram | 526 | 0.6 | 4 | 3.6 | 15 |
| GSL Matrix | 7233 | 16.9 | 8 | 1.8 | 7 |

# Second Experiment

We also verified a complicated assertion each on five real programs

| Name | LOC | Time (s) | # queries | Avg # vars in query | Avg LOC in query |
|---|---|---|---|---|---|
| Wizardpen Linux Driver | 1242 | 3.8 | 5 | 1.5 | 29 |
| OpenSSH clientloop | 1987 | 2.8 | 3 | 2.3 | 5 |
| Coreutils su | 1057 | 3.0 | 5 | 1.7 | 6 |
| GSL Histogram | 526 | 0.6 | 4 | 3.6 | 15 |
| GSL Matrix | 7233 | 16.9 | 8 | 1.8 | 7 |

**Fragments extracted for queries small in practice**

## Summary

- New technique to decompose program's correctness proof into small lemmas

## Summary

- New technique to decompose program's correctness proof into small lemmas
- Allows a portfolio of diverse analyses with different strengths to cooperate

## Summary

- New technique to decompose program's correctness proof into small lemmas
- Allows a portfolio of diverse analyses with different strengths to cooperate
- Interaction is goal-directed and effective in practice

## Summary

- New technique to decompose program's correctness proof into small lemmas
- Allows a portfolio of diverse analyses with different strengths to cooperate
- Interaction is goal-directed and effective in practice