Automated Error Diagnosis Using Abductive Inference

Işıl Dillig, Tom Dillig
College of William & Mary

Alex Aiken
Stanford University

Motivation

- If we use sound program analysis tool to verify a property, answer is either yes or no

Motivation

- If we use sound program analysis tool to verify a property, answer is either yes or no
- If answer is yes, program is error-free

Motivation

- If we use sound program analysis tool to verify a property, answer is either yes or no
- If answer is yes, program is error-free
- If answer is no, there are two possibilities:

Motivation

- If we use sound program analysis tool to verify a property, answer is either yes or no
- If answer is yes, program is error-free
- If answer is no, there are two possibilities:
    - Either the program is indeed buggy

Motivation

- If we use sound program analysis tool to verify a property, answer is either yes or no
- If answer is yes, program is error-free
- If answer is no, there are two possibilities:
  - Either the program is indeed buggy
  - Or report is a false alarm

When Verification Fails

- When verifier fails to prove property, user must decide whether report is real bug or false alarm.

When Verification Fails

- When verifier fails to prove property, user must decide whether report is real bug or false alarm.
- But manually classifying error reports is time-consuming and error-prone.

When Verification Fails

- When verifier fails to prove property, user must decide whether report is real bug or false alarm.
- But manually classifying error reports is time-consuming and error-prone.
- Furthermore, user must redo all the reasoning the tool performed just to discover where it became stuck.

When Verification Fails

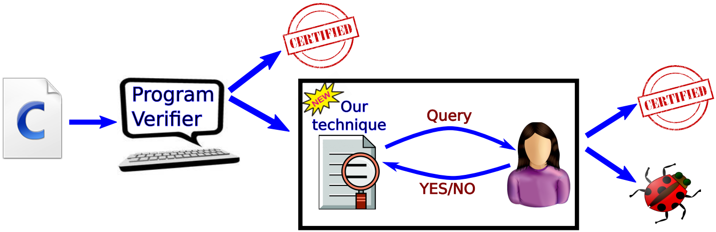
- When verifier fails to prove property, user must decide whether report is real bug or false alarm.
- But manually classifying error reports is time-consuming and error-prone.
- Furthermore, user must redo all the reasoning the tool performed just to discover where it became stuck.
- Very painful process for most users of static analysis tools!

Our Goal

A new technique for semi-automating error report classification when automated program verification fails

Our Goal

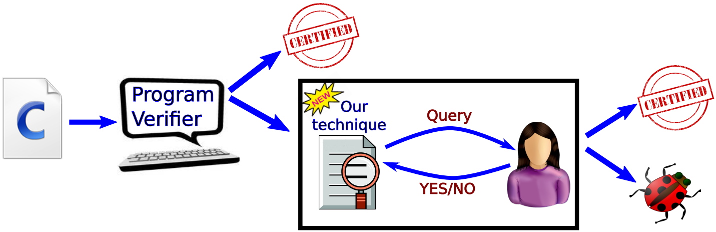
A new technique for semi-automating error report classification when automated program verification fails



- Allows verifier to interact with user by asking small, relevant queries until report is classified as real bug or false positive

Our Goal

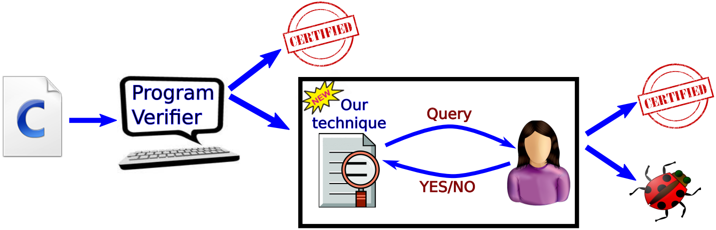
A new technique for semi-automating error report classification when automated program verification fails



- Allows verifier to interact with user by asking small, relevant queries until report is classified as real bug or false positive
- Queries capture only the information verifier is missing $\Rightarrow$ user contributes facts verifier could not decide on its own

Our Goal

A new technique for semi-automating error report classification when automated program verification fails



- Allows verifier to interact with user by asking small, relevant queries until report is classified as real bug or false positive
- Queries capture only the information verifier is missing $\Rightarrow$ user contributes facts verifier could not decide on its own
- Answering queries much easier than classifying error report

Key Ideas

Key Idea #1: Analysis makes explicit not only facts it knows, but also facts it does not know

Key Ideas

Key Idea #1: Analysis makes explicit not only facts it knows, but also facts it does not know

- Sources of imprecision/incompleteness in static analysis represented using abstraction variables

Key Idea #1: Analysis makes explicit not only facts it knows, but also facts it does not know

- Sources of imprecision/incompleteness in static analysis represented using abstraction variables
- For example, if value of variable is unknown after a loop, represent this unknown value using abstraction variable

Key Ideas

Key Idea #1: Analysis makes explicit not only facts it knows, but also facts it does not know

- Sources of imprecision/incompleteness in static analysis represented using abstraction variables
- For example, if value of variable is unknown after a loop, represent this unknown value using abstraction variable
- This representation allows analysis to be "introspective" and reason about what facts it could be missing

Key Ideas, cont.

Key Idea #2: Abductive inference

Key Ideas, cont.

Key Idea #2: Abductive inference

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

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

Key Ideas, cont.

Key Idea #2: Abductive inference

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

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

- We use abductive inference to generate simple explanations that either guarantee that program is error-free or definitely buggy

Key Ideas, cont.

Key Idea #2: Abductive inference

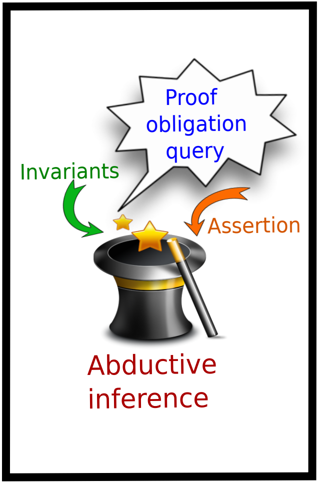
- Given known facts F and desired outcome O, abductive inference finds simple explanatory hypothesis E such that

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

- We use abductive inference to generate simple explanations that either guarantee that program is error-free or definitely buggy
- These abductive explanations are presented as queries to user

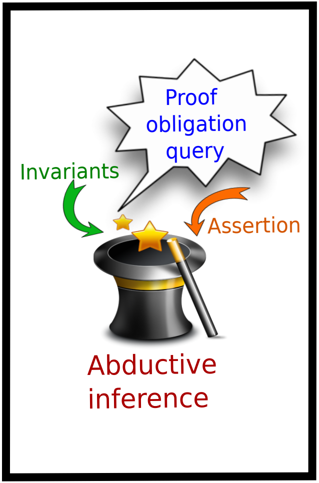
Proof Obligation via Abductive Inference

- Input: invariants computed by verifier and assertion to discharge



Proof Obligation via Abductive Inference

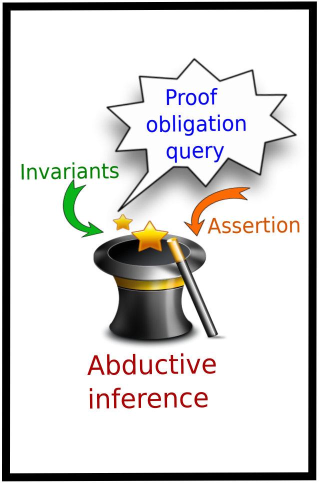
- Input: invariants computed by verifier and assertion to discharge
- Technique computes formulas I and ϕ describing invariant and assertion in terms of abstraction variables



Proof Obligation via Abductive Inference

- Input: invariants computed by verifier and assertion to discharge
- Technique computes formulas I and ϕ describing invariant and assertion in terms of abstraction variables
- Use abduction to compute simple and general explanation Γ s.t.:

$$\Gamma \wedge I \models \phi \quad \text{and} \quad \text{SAT}(\Gamma \wedge I)$$

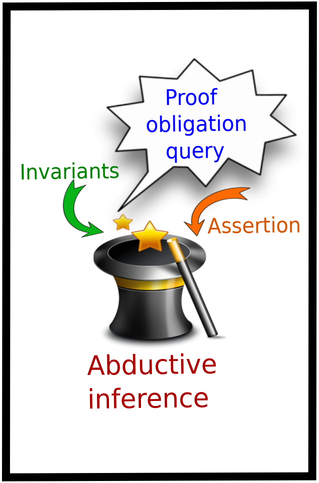


Proof Obligation via Abductive Inference

- Input: invariants computed by verifier and assertion to discharge
- Technique computes formulas I and ϕ describing invariant and assertion in terms of abstraction variables
- Use abduction to compute simple and general explanation Γ s.t.:

$$\Gamma \wedge I \models \phi \quad \text{and} \quad \text{SAT}(\Gamma \wedge I)$$

- Abductive explanation Γ is presented to user as proof obligation query

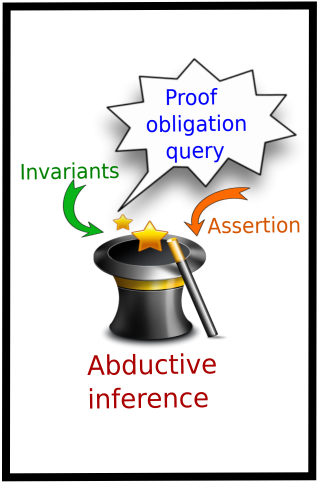


Proof Obligation via Abductive Inference

- Input: invariants computed by verifier and assertion to discharge
- Technique computes formulas I and ϕ describing invariant and assertion in terms of abstraction variables
- Use abduction to compute simple and general explanation Γ s.t.:

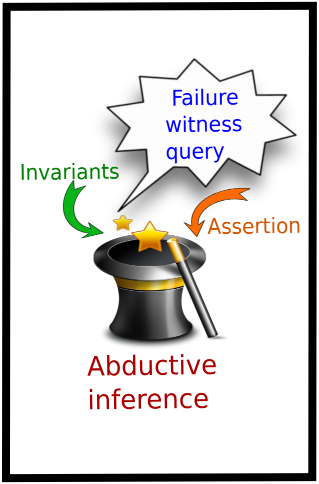
$$\Gamma \wedge I \models \phi \quad \text{and} \quad \text{SAT}(\Gamma \wedge I)$$

- Abductive explanation Γ is presented to user as proof obligation query
- If Γ is invariant, report is false alarm



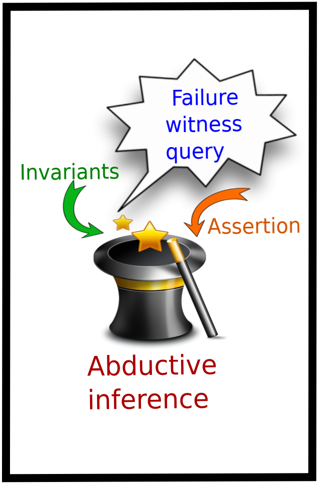
Failure Witnesses

- Proof obligation query used to show report is false alarm



Failure Witnesses

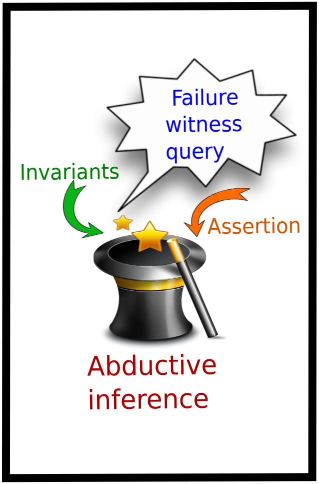
- Proof obligation query used to show report is false alarm
- We generate another query, called failure witness query, to show report is a real bug



Failure Witnesses

- Proof obligation query used to show report is false alarm
- We generate another query, called failure witness query, to show report is a real bug
- To generate failure witness query, solve a dual abductive inference problem:

$$\Delta \land I \models \neg\phi \quad \text{and} \quad \mathrm{SAT}(\Delta \land I)$$

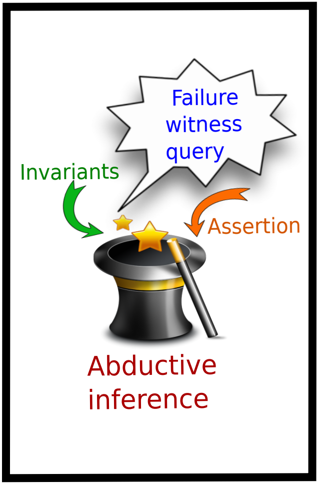


Failure Witnesses

- Proof obligation query used to show report is false alarm
- We generate another query, called failure witness query, to show report is a real bug
- To generate failure witness query, solve a dual abductive inference problem:

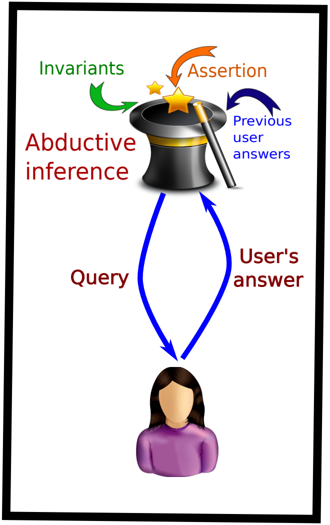
$$\Delta \wedge I \models \neg\phi \quad \text{and} \quad \mathrm{SAT}(\Delta \wedge I)$$

- If Δ can hold in some program execution, then report is real bug!



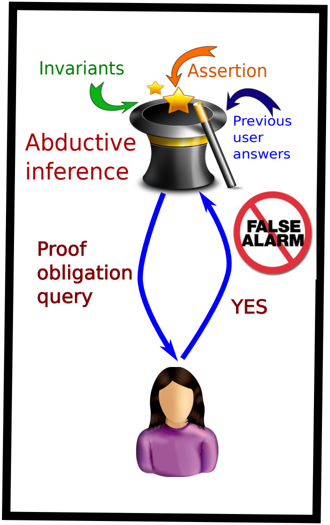
Automated Error Diagnosis via Abductive Inference

- Our technique helps user classify error reports by generating simple queries



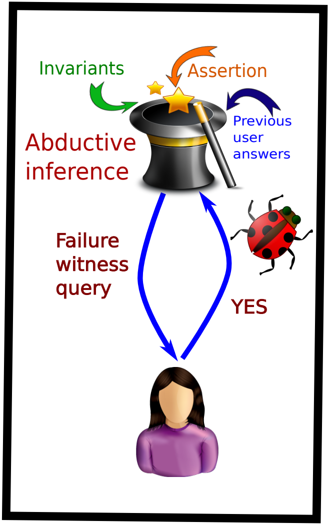
Automated Error Diagnosis via Abductive Inference

- Our technique helps user classify error reports by generating simple queries
- If query is a proof obligation and user answers yes, report classified as false alarm



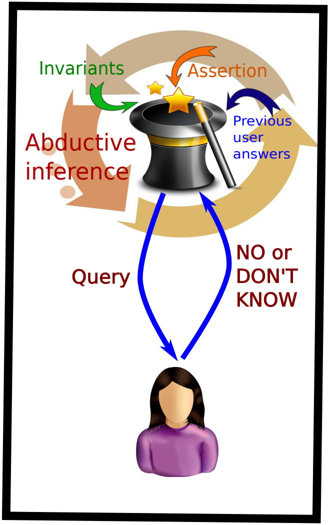
Automated Error Diagnosis via Abductive Inference

- Our technique helps user classify error reports by generating simple queries
- If query is a proof obligation and user answers yes, report classified as false alarm
- If query is a failure witness and user answers yes, report classified as real bug



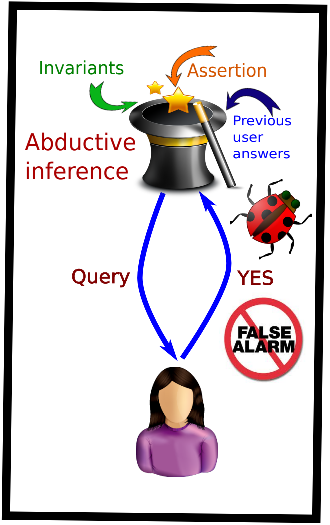
Automated Error Diagnosis via Abductive Inference

- Our technique helps user classify error reports by generating simple queries
- If query is a proof obligation and user answers yes, report classified as false alarm
- If query is a failure witness and user answers yes, report classified as real bug
- If user answers "no" or "I don't know", technique computes new abductive explanation distinct from previous ones



Automated Error Diagnosis via Abductive Inference

- Our technique helps user classify error reports by generating simple queries
- If query is a proof obligation and user answers yes, report classified as false alarm
- If query is a failure witness and user answers yes, report classified as real bug
- If user answers "no" or "I don't know", technique computes new abductive explanation distinct from previous ones
- Interaction continues until report is classified as real bug or false alarm



Example

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Suppose a verification tool reports potential error for this example

Example

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Suppose a verification tool reports potential error for this example
- Want to classify report as false alarm or real bug using our technique

Example

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Suppose a verification tool reports potential error for this example
- Want to classify report as false alarm or real bug using our technique
- First, perform symbolic value flow analysis, representing each unknown value as an abstraction variable α

Example

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Suppose a verification tool reports potential error for this example
- Want to classify report as false alarm or real bug using our technique
- First, perform symbolic value flow analysis, representing each unknown value as an abstraction variable α
- Since precise values of `i` and `j` are unknown after loop, represent their values using α_i and α_j

Example

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Suppose a verification tool reports potential error for this example
- Want to classify report as false alarm or real bug using our technique
- First, perform symbolic value flow analysis, representing each unknown value as an abstraction variable α
- Since precise values of `i` and `j` are unknown after loop, represent their values using α_i and α_j
- Similarly, represent unknown value of `x` as abstraction variable α_x

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Perform symbolic value propagation to represent `z`'s value in terms of α's and function inputs

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Perform symbolic value propagation to represent `z`'s value in terms of α's and function inputs
- If `flag` is zero, `z` $= 1 + \alpha_i + \alpha_j$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Perform symbolic value propagation to represent `z`'s value in terms of α's and function inputs
- If `flag` is zero, `z` $= 1 + \alpha_i + \alpha_j$
- If `flag` is non-zero, `z` $= \alpha_x + \alpha_i + \alpha_j$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Perform symbolic value propagation to represent z's value in terms of α's and function inputs
- If flag is zero, z $= 1 + \alpha_i + \alpha_j$
- If flag is non-zero, z $= \alpha_x + \alpha_i + \alpha_j$
- Thus, condition under which assertion succeeds is:

$$\phi = \begin{array}{l} (1 + \alpha_i + \alpha_j > 2 * n \wedge \neg flag) \vee \\ (\alpha_x + \alpha_i + \alpha_j > 2 * n \wedge flag) \end{array}$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Now, we want to utilize invariants inferred by verification tool

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Now, we want to utilize invariants inferred by verification tool
- Suppose verifier inferred `havoc` returns non-negative value: $\alpha_x \geq 0$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Now, we want to utilize invariants inferred by verification tool
- Suppose verifier inferred `havoc` returns non-negative value: $\alpha_x \geq 0$
- And that `i` is greater than `n` after loop: $\alpha_i > n$

Example, cont.

```
void foo(int flag,
          unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Now, we want to utilize invariants inferred by verification tool
- Suppose verifier inferred `havoc` returns non-negative value: $\alpha_x \geq 0$
- And that `i` is greater than `n` after loop: $\alpha_i > n$
- Finally, since `n` is unsigned, $n \geq 0$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Now, we want to utilize invariants inferred by verification tool
- Suppose verifier inferred `havoc` returns non-negative value: $\alpha_x \geq 0$
- And that `i` is greater than `n` after loop: $\alpha_i > n$
- Finally, since `n` is unsigned, $n \geq 0$
- Putting this all together, we know the invariants:

$$\mathcal{I} = \alpha_x \geq 0 \land \alpha_i > 0 \land n \geq 0$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

$$\phi = \begin{array}{l}(1+\alpha_i+\alpha_j > 2*n \wedge \neg flag) \vee \\ (\alpha_x+\alpha_i+\alpha_j > 2*n \wedge flag)\end{array}$$

$$\mathcal{I} \;=\; \alpha_x \geq 0 \wedge \alpha_i > 0 \wedge n \geq 0$$

- To classify error report, we solve two abductive inference problems.

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

$$\phi = \begin{array}{l}(1 + \alpha_i + \alpha_j > 2 * n \land \neg flag) \lor \\ (\alpha_x + \alpha_i + \alpha_j > 2 * n \land flag)\end{array}$$

$$\mathcal{I} \;=\; \alpha_x \geq 0 \land \alpha_i > 0 \land n \geq 0$$

- To classify error report, we solve two abductive inference problems.
- First, find proof obligation Γ s.t:

$$\Gamma \land \mathcal{I} \models \phi$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

$$\phi = \begin{array}{l} (1+\alpha_i+\alpha_j > 2*n \wedge \neg flag) \vee \\ (\alpha_x+\alpha_i+\alpha_j > 2*n \wedge flag) \end{array}$$

$$\mathcal{I} \; = \; \alpha_x \geq 0 \wedge \alpha_i > 0 \wedge n \geq 0$$

- To classify error report, we solve two abductive inference problems.
- First, find proof obligation Γ s.t:

$$\Gamma \wedge \mathcal{I} \models \phi$$

- Solution computed by our technique is:

$$\Gamma = \; \alpha_j \geq n$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

$$\phi = \begin{array}{l}(1 + \alpha_i + \alpha_j > 2 * n \land \neg flag) \lor \\ (\alpha_x + \alpha_i + \alpha_j > 2 * n \land flag)\end{array}$$

$$\mathcal{I} \;=\; \alpha_x \geq 0 \land \alpha_i > 0 \land n \geq 0$$

- Next, solve another abductive inf. problem to compute failure witness Δ:

$$\Delta \land \mathcal{I} \models \neg\phi$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

$$\phi = \begin{array}{l}(1+\alpha_i+\alpha_j > 2*n \wedge \neg flag) \vee \\ (\alpha_x+\alpha_i+\alpha_j > 2*n \wedge flag)\end{array}$$

$$\mathcal{I} \;=\; \alpha_x \geq 0 \wedge \alpha_i > 0 \wedge n \geq 0$$

- Next, solve another abductive inf. problem to compute failure witness Δ:

$$\Delta \wedge \mathcal{I} \models \neg\phi$$

- Solution computed by our technique is:

$$\Delta = \;\neg flag \wedge \alpha_i + \alpha_j < 0$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }

  int z = k+i+j;
  assert(z>2*n);
}
```

- Next, we compare Γ and Δ to decide which one is more promising:

$$\Gamma = \alpha_j \geq n \quad \Delta = \neg \mathit{flag} \wedge \alpha_i + \alpha_j < 0$$

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }
 Query: Is j>=n invariant?
  int z = k+i+j;
  assert(z>2*n);
}
```

- Next, we compare Γ and Δ to decide which one is more promising:

  $\Gamma = \alpha_j \geq n \quad \Delta = \neg \mathit{flag} \wedge \alpha_i + \alpha_j < 0$

- Technique decides Γ more promising, thus we query user if `j >= n`

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }
 Query: Is j>=n invariant?
  int z = k+i+j;
  assert(z>2*n);
}
```

- Next, we compare Γ and Δ to decide which one is more promising:

  $\Gamma = \alpha_j \geq n \quad \Delta = \neg flag \wedge \alpha_i + \alpha_j < 0$

- Technique decides Γ more promising, thus we query user if `j >= n`
- In this case, easy to show `j >= n` is invariant

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }
```

Query: Is j>=n invariant?

```
  int z = k+i+j;
  assert(z>2*n);
}
```

- Next, we compare Γ and Δ to decide which one is more promising:

  $$\Gamma = \alpha_j \geq n \quad \Delta = \neg \mathit{flag} \wedge \alpha_i + \alpha_j < 0$$

- Technique decides Γ more promising, thus we query user if `j >= n`
- In this case, easy to show `j >= n` is invariant
- Thus, we classify report as false alarm

Example, cont.

```
void foo(int flag,
         unsigned int n)
{
  int k = 1;
  int x = havoc();
  if(flag) k = x;

  int i=0, j=0;
  while(i<=n)
  {
    i++;
    j+=i;
  }
```

Query: Is j>=n invariant?

```
  int z = k+i+j;
  assert(z>2*n);
}
```

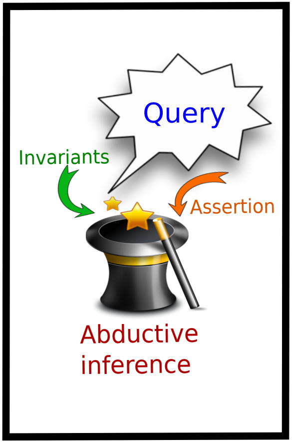
- Next, we compare Γ and Δ to decide which one is more promising:

  $\Gamma = \alpha_j \geq n \quad \Delta = \neg flag \wedge \alpha_i + \alpha_j < 0$

- Technique decides Γ more promising, thus we query user if `j >= n`
- In this case, easy to show `j >= n` is invariant
- Thus, we classify report as false alarm
- Easier to answer this query than to manually classify error report

Computing Abductive Explanations

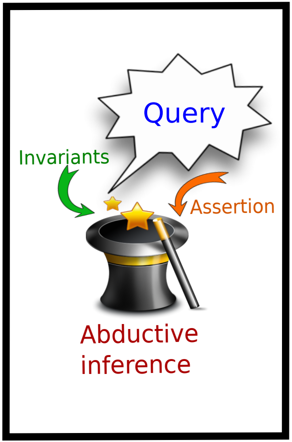
- Abduction is useful, but how do we compute these explanations?



Computing Abductive Explanations

- Abduction is useful, but how do we compute these explanations?
- Given invariants I and desired outcome ϕ, how to find explanation E s.t.:

$$I \land E \models \phi \quad \land \quad \mathrm{SAT}(I \land E)$$

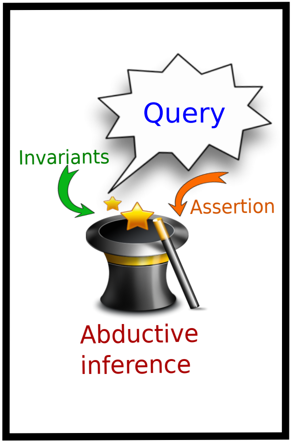


Computing Abductive Explanations

- Abduction is useful, but how do we compute these explanations?
- Given invariants I and desired outcome ϕ, how to find explanation E s.t.:

$$I \land E \models \phi \quad \land \quad \mathrm{SAT}(I \land E)$$

- Trivial solution is $E = \phi$, but useless b/c same as asking user to prove assertion!

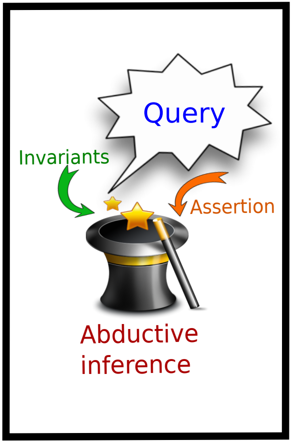


Computing Abductive Explanations

- Abduction is useful, but how do we compute these explanations?
- Given invariants I and desired outcome ϕ, how to find explanation E s.t.:

$$I \land E \models \phi \quad \land \quad \mathrm{SAT}(I \land E)$$

- Trivial solution is $E = \phi$, but useless b/c same as asking user to prove assertion!
- Want solutions that are as simple and as general as possible!

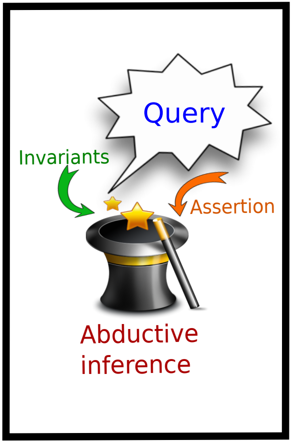


Computing Abductive Explanations

- Abduction is useful, but how do we compute these explanations?
- Given invariants I and desired outcome ϕ, how to find explanation E s.t.:

$$I \land E \models \phi \quad \land \quad \mathrm{SAT}(I \land E)$$

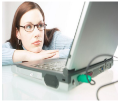
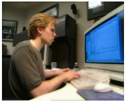
- Trivial solution is $E = \phi$, but useless b/c same as asking user to prove assertion!
- Want solutions that are as simple and as general as possible!



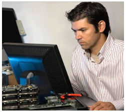
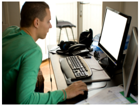
Use minimum satisfying assignments and quantifier elimination to compute simple and general explanations

Experimental Evaluation

- Performed user study to evaluate new technique

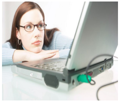
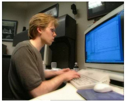



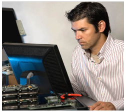
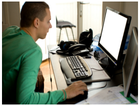
oDesk


Experimental Evaluation

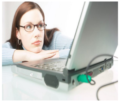
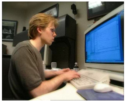
- Performed user study to evaluate new technique
- Hired 56 programmers through ODesk and asked them to classify error reports

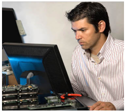
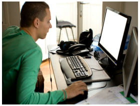




Experimental Evaluation

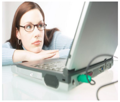
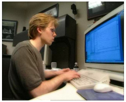
- Performed user study to evaluate new technique
- Hired 56 programmers through ODesk and asked them to classify error reports
- Each programmer asked to classify (randomly selected) half of reports manually, and other half using our technique

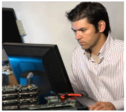
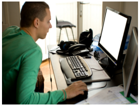




Experimental Evaluation

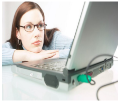
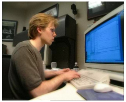
- Performed user study to evaluate new technique
- Hired 56 programmers through ODesk and asked them to classify error reports
- Each programmer asked to classify (randomly selected) half of reports manually, and other half using our technique
- Manual classification: Given code and error report, decide if bug, false alarm, or unknown

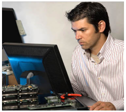
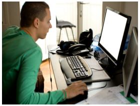




Experimental Evaluation

- Performed user study to evaluate new technique
- Hired 56 programmers through ODesk and asked them to classify error reports
- Each programmer asked to classify (randomly selected) half of reports manually, and other half using our technique
- Manual classification: Given code and error report, decide if bug, false alarm, or unknown
- Our technique: Given code and series of queries, asked to answer “Yes”, “No”, or “Don’t know”

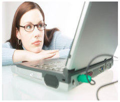
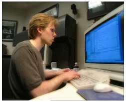



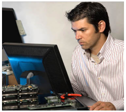
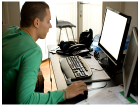
oDesk


Experimental Evaluation

- Performed user study to evaluate new technique
- Hired 56 programmers through ODesk and asked them to classify error reports
- Each programmer asked to classify (randomly selected) half of reports manually, and other half using our technique
- Manual classification: Given code and error report, decide if bug, false alarm, or unknown
- Our technique: Given code and series of queries, asked to answer “Yes”, “No”, or “Don’t know”
- Based on answers to queries, report classified automatically

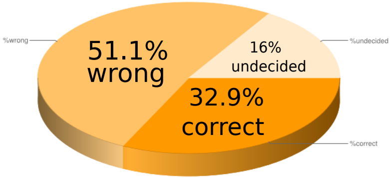




Results of User Study

- With manual classification, programmers classified 51.1% of reports incorrectly

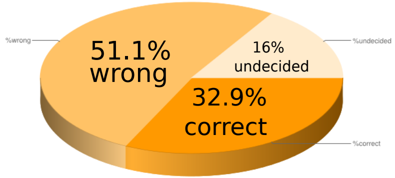
Manual Classification



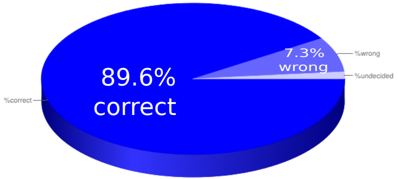
Results of User Study

- With manual classification, programmers classified 51.1% of reports incorrectly
- With assisted classification, programmers classified only 7.3% of reports incorrectly

Manual Classification



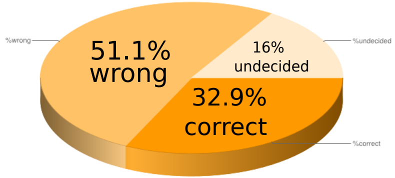
Assisted Classification



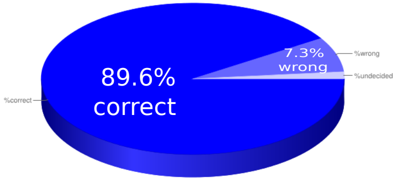
Results of User Study

- With manual classification, programmers classified 51.1% of reports incorrectly
- With assisted classification, programmers classified only 7.3% of reports incorrectly
- Our technique dramatically improves classification accuracy

Manual Classification



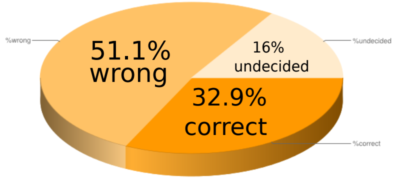
Assisted Classification



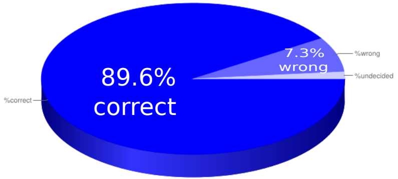
Results of User Study

- With manual classification, programmers classified 51.1% of reports incorrectly
- With assisted classification, programmers classified only 7.3% of reports incorrectly
- Our technique dramatically improves classification accuracy
- Also dramatically reduces time needed to classify report

Manual Classification



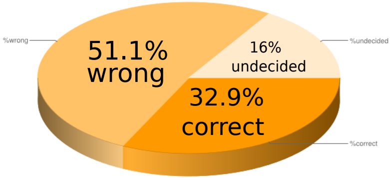
Assisted Classification



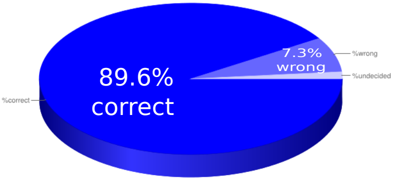
Results of User Study

- With manual classification, programmers classified 51.1% of reports incorrectly
- With assisted classification, programmers classified only 7.3% of reports incorrectly
- Our technique dramatically improves classification accuracy
- Also dramatically reduces time needed to classify report
- Using manual classification, programmers need 293 seconds on average

Manual Classification



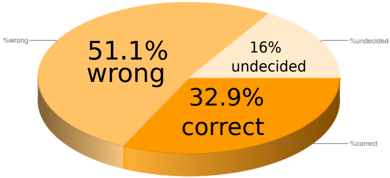
Assisted Classification



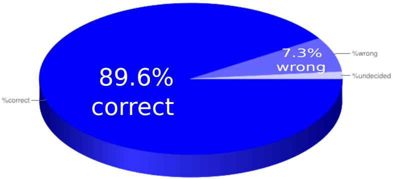
Results of User Study

- With manual classification, programmers classified 51.1% of reports incorrectly
- With assisted classification, programmers classified only 7.3% of reports incorrectly
- Our technique dramatically improves classification accuracy
- Also dramatically reduces time needed to classify report
- Using manual classification, programmers need 293 seconds on average
- Using new technique, programmers take 55 seconds on average

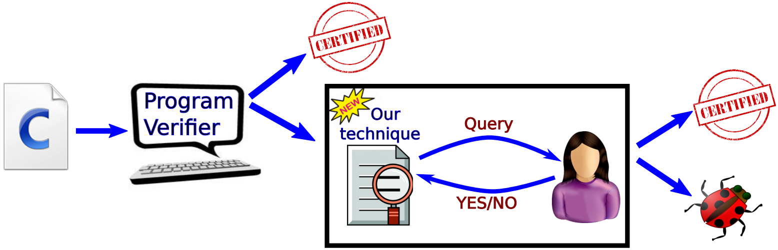
Manual Classification



Assisted Classification

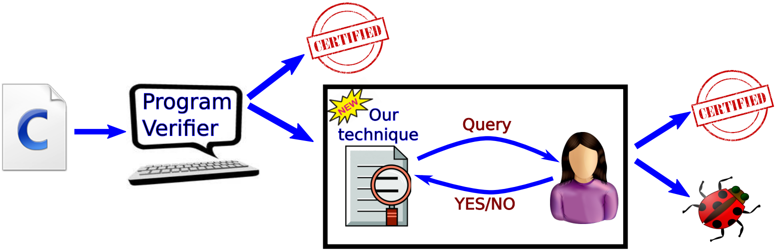


Summary



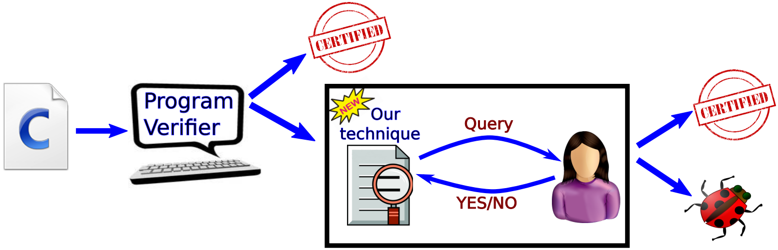
- New technique to help programmers classify error reports as real bugs or false alarms

Summary



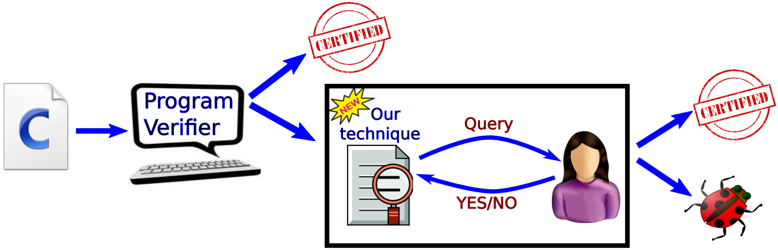
- New technique to help programmers classify error reports as real bugs or false alarms
- Uses abductive inference to compute simple queries that capture what analysis is missing

Summary



- New technique to help programmers classify error reports as real bugs or false alarms
- Uses abductive inference to compute simple queries that capture what analysis is missing
- Interacts with user until report is classified as bug/false alarm

Summary



- New technique to help programmers classify error reports as real bugs or false alarms
- Uses abductive inference to compute simple queries that capture what analysis is missing
- Interacts with user until report is classified as bug/false alarm
- User study shows technique dramatically improves classification speed and accuracy

Related Work:

- Ball, T., Naik, M., Rajamani, S.: From Symptom to Cause: Localizing Errors in Counterexample Traces. POPL 2003
- Jose, M., Majumdar, R.: Cause Clue Clauses: Error Localization using Maximum Satisfiability. PLDI 2011
- Groce, A.: Error Explanation with Distance Metrics. TACAS 2004
- Dillig, I., Dillig, T., McMillan, K., Aiken, A.: Minimum Satisfying Assignments for SMT. CAV 2012.