Automatic Generation of Local Repairs for Boolean Programs

Roopsha Samanta,
Jyotirmoy V. Deshmukh and E. Allen Emerson

The University of Texas at Austin

April 5, 2012

Outline

- Motivation
- Solution Framework
- The Algorithm
- Conclusions


End Sub
Private Sub tbToolBar_
On Error Resume Next
timTimer.Enabled = True
Select Case Button.Key
Case "Back"
brwWebBrowser.
Case
Case
brwWebBrows
Case "Home"
Case

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual debugging

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual *debugging*

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual *debugging*

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual *debugging*

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual *debugging*

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual *debugging*

The road to correct programs . . .

- Program *synthesis*
  - Correct by construction
  - Detailed specification
  - Hard
  - Also, legacy code?
- Program *verification*
- Program design + *verification* + *fault localization* + *repair*
  - Lengthy, iterative cycle
  - Long, unreadable error traces
  - Essentially manual *debugging*

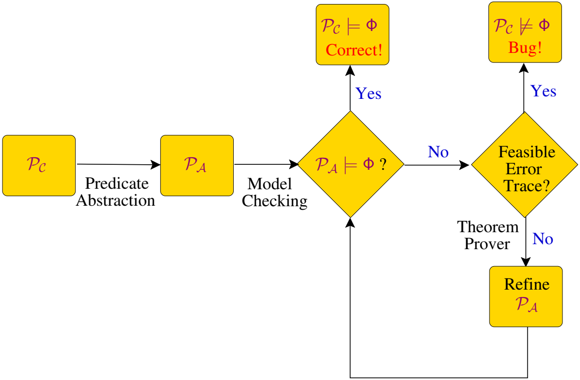
The repair problem

Given a program $\mathcal{P}$ and a specification Φ such that $\mathcal{P} \not\models \Phi$, transform $\mathcal{P}$ to $\mathcal{P}'$ such that $\mathcal{P}' \models \Phi$

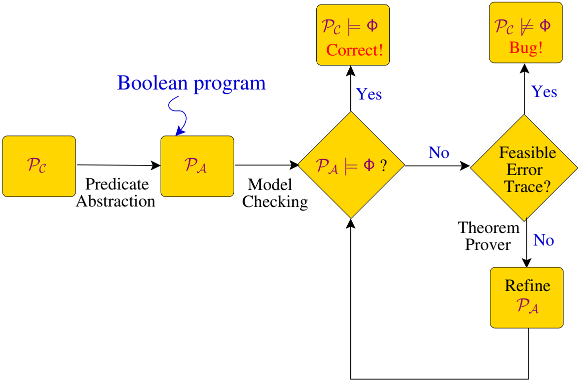
A specialization . . .

- Program model: sequential Boolean programs
- Specifications: Hoare-style pre-conditions, post-conditions
- Permissible faults/repairs: incorrect Boolean expressions

Iterative (predicate) abstraction-refinement



Iterative (predicate) abstraction-refinement



What are Boolean programs?

- Abstractions of concrete programs
- Boolean variables
- Similar control flow
  - Conditionals, loops, procedures
- Nondeterminism
  - Some expressions may evaluate to either *true* or *false*

Example C program and Boolean program

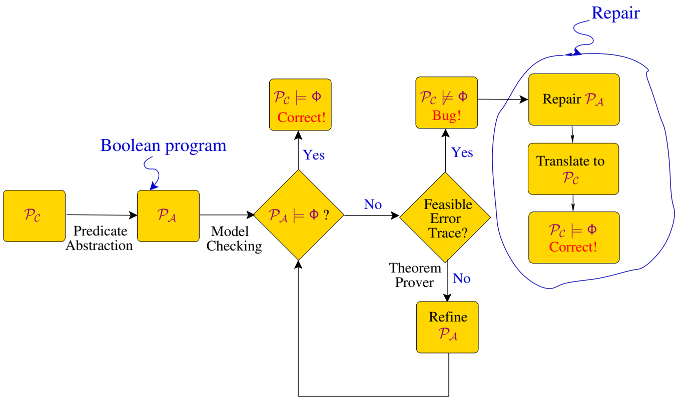
```
while (x>0){
  x := x-1;
}
```

```
while (p){
  p := nd(0,1);
}
```

Why Boolean programs?

- Used as program abstractions for software verification
  - *e.g.*, SLAM, BLAST, *etc.*

Repair of software programs



Why Boolean programs?

- Used as program abstractions for software verification
  - *e.g.*, SLAM, BLAST, *etc.*
- Could be used to model some Boolean circuits

Program Syntax

- Prog $\mathcal{P} = (\mathcal{V}, \mathtt{main}, \mathcal{F})$
  - $\mathcal{V} = \{v_1, v_2, \ldots, v_t\}$: Boolean vars
  - $\mathtt{main} = (S, \mathcal{V})$, S: $s_1; s_2; \ldots; s_n$: stmts
  - $\mathcal{F}$: functions, $f = (S_f, \mathcal{V}_{f,l})$
- Expr E: Boolean expr + $nd(0,1)$
  - *e.g.*, $v_2 \wedge nd(0,1)$
- Prog stmt s_i: function call or `return` or,
  - assignment: v_j `:=` E`;`
  - conditional: `if (`G`)` S_{if} `else` S_{else}`;`
  - loop: `while (`G`)` S_{body}`;`

Program Syntax

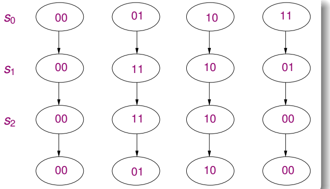
- Prog $\mathcal{P} = (\mathcal{V}, \texttt{main}, \mathcal{F})$
  - $\mathcal{V} = \{v_1, v_2, \ldots, v_t\}$: Boolean vars
  - $\texttt{main} = (S, \mathcal{V})$, S: $s_1; s_2; \ldots; s_n$: stmts
  - $\mathcal{F}$: functions, $f = (S_f, \mathcal{V}_{f,l})$
- Expr E: Boolean expr + $nd(0, 1)$
  - *e.g.*, $v_2 \wedge nd(0, 1)$
- Prog stmt s_i: function call or `return` or,
  - assignment: v_j `:=` E`;`
  - conditional: `if` `(`G`)` S_{if} `else` S_{else}`;`
  - loop: `while` `(`G`)` S_{body}`;`

Program Syntax

- Prog $\mathcal{P} = (\mathcal{V}, \texttt{main}, \mathcal{F})$
  - $\mathcal{V} = \{v_1, v_2, \ldots, v_t\}$: Boolean vars
  - $\texttt{main} = (S, \mathcal{V})$, S: $s_1; s_2; \ldots; s_n$: stmts
  - $\mathcal{F}$: functions, $f = (S_f, \mathcal{V}_{f,l})$
- Expr E: Boolean expr + $nd(0, 1)$
  - *e.g.*, $v_2 \wedge nd(0, 1)$
- Prog stmt s_i: function call or `return` or,
  - assignment: v_j `:=` E`;`
  - conditional: `if (`G`)` S_{if} `else` S_{else}`;`
  - loop: `while (`G`)` S_{body}`;`

Example Boolean program and its state diagram

```
swap(x,y){
   x := x ⊕ y;
   y := x ∧ y;
   x := x ⊕ y;
}
```



Specification

Total correctness: $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$

- Pre-condition φ : init states of $\mathcal{P}$
- Post-condition ψ : desired final states

$\mathcal{P}$ is correct *iff* execution of $\mathcal{P}$, begun in any state in φ, terminates in a state in ψ, for *all* choices that $\mathcal{P}$ might make.

Specification

Total correctness: $\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$

- Pre-condition φ : init states of $\mathcal{P}$
- Post-condition ψ : desired final states

$\mathcal{P}$ is correct *iff* execution of $\mathcal{P}$, begun in any state in φ, terminates in a state in ψ, for *all* choices that $\mathcal{P}$ might make.

Example Boolean program with its specification

φ : *true*

```
x := x ⊕ y;
y := x ∧ y;
x := x ⊕ y;
```

$$\psi : (y_f \equiv x(0) \wedge (x(f) \equiv y(0))$$

Fault/repair model

- Extra statement (needs deletion)
- Assignment: faulty LHS or RHS
- Conditional: faulty G or faulty statement in S_{if} or S_{else}
- Loop: faulty G or faulty statement in S_{body}

Our algorithm seeks to repair only the above kinds of faults.

Fault/repair model

- Extra statement (needs deletion)
- Assignment: faulty LHS or RHS
- Conditional: faulty G or faulty statement in S_{if} or S_{else}
- Loop: faulty G or faulty statement in S_{body}

Our algorithm seeks to repair only the above kinds of faults.

Algorithm sketch

- *Annotation:*
  - Propagate φ and ψ through statements
- *Repair:*
  - Use annotations to inspect statements for *repairability*
  - Generate repair if possible

Program annotation

φ_0 : *true*

Incorrect Program

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Program annotation

Incorrect Program

φ_0 : *true*

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Post-condition propagation

Program annotation

φ_0 : *true*

Incorrect Program

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

ψ_2

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Post-condition propagation

Program annotation

φ_0 : *true*

Incorrect Program

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

ψ_1

ψ_2

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Post-condition propagation

Program annotation

φ_0 : *true*

Incorrect Program

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

ψ_0

ψ_1

ψ_2

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Post-condition propagation

Program annotation

Pre-condition propagation

φ_0 : *true*

Incorrect Program

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

ψ_0

ψ_1

ψ_2

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Post-condition propagation

Program annotation

Pre-condition propagation

φ_0 : *true*

φ_1

φ_2

φ_3

Incorrect Program

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

ψ_0

ψ_1

ψ_2

$\psi_3 : x(f) \equiv y(0) \wedge y(f) \equiv x(0)$

Post-condition propagation

Backward propagation of ψ_i through s_i

Weakest pre-condition $wp(s_i, \psi_i)$:
Set of all *input* states from which s_i is guaranteed to terminate in ψ_i for all choices made by s_i.

To propagate ψ_i back through s_i, compute $wp(s_i, \psi_i)$.

Details . . .

Assignments: $v_j \ := \ E$;
$\psi_{i-1} = \psi_i[v'_j \rightarrow E$, for each $\ m \neq j, v'_m \rightarrow v_m]$

Rule for sequential composition:
$wp((s_{i-1}; s_i), \psi_i) = wp(s_{i-1}, wp(s_i, \psi_i))$

Conditionals: `if (`G`)` S_{if} `else` S_{else};
$\psi_{i-1} = (G \Rightarrow wp(S_{if}, \psi_i)) \ \wedge \ (\neg G \Rightarrow wp(S_{else}, \psi_i))$

Loops: `while (`G`)` S_{body};
$\psi_{i-1} \ = \ (\psi_i \wedge \neg G) \vee \bigvee_{l=1}^{L} wp(S_{body}, Y_{l-1} \wedge \neg G)$
where, $Y_0 = \psi_i$, $Y_k = wp(S_{body}, Y_{k-1} \wedge \neg G)$

Forward propagation of φ_{i-1} through s_i

Strongest post-condition $sp(s_i, \varphi_{i-1})$:
Smallest set of *output* states in which s_i is guaranteed to terminate, starting in φ_{i-1}, for all choices that s_i might make.

To propagate φ_{i-1} forward through s_i, compute $sp(s_i, \varphi_{i-1})$.

Example program annotation

Pre-condition propagation

φ_0: $true$

φ_1: $x' \equiv (x(0) \oplus y(0)) \wedge$
$y' \equiv y(0)$

φ_2: $x' \equiv (x(0) \oplus y(0)) \wedge$
$y' \equiv (\neg x(0) \wedge y(0))$

φ_3: $x' \equiv (x(0) \wedge \neg y(0)) \wedge$
$y' \equiv (\neg x(0) \wedge y(0))$

Incorrect Program

```
x' := x(0) ⊕ y(0);

y' := x ∧ y;

x(f) := x ⊕ y;
```

ψ_0: $y(0) \equiv (x(0) \wedge \neg y(0)) \wedge$
$x(0) \equiv (\neg x(0) \wedge y(0))$

ψ_1: $y(0) \equiv (x \wedge \neg y) \wedge$
$x(0) \equiv (x \wedge y)$

ψ_2: $y(0) \equiv x \oplus y \wedge$
$x(0) \equiv y$

ψ_3: $x(f) \equiv y(0) \wedge$
$y(f) \equiv x(0)$

Post-condition propagation

Local Hoare triples

φ_0 φ_1 φ_2 φ_3

s_0: `x' := x(0)` $\oplus$ `y(0);`

s_1: `y' := x` $\wedge$ `y;`

s_2: `x(f) := x` $\oplus$ `y;`

ψ_0 ψ_1 ψ_2 ψ_3

Local Hoare triples

Local Hoare triple: $\langle \varphi_0 \rangle s_0 \langle \psi_1 \rangle$

φ_0 ψ_0

s_0: `x' := x(0)` $\oplus$ `y(0);`

φ_1 ψ_1

s_1: `y' := x` $\wedge$ `y;`

φ_2 ψ_2

s_2: `x(f) := x` $\oplus$ `y;`

φ_3 ψ_3

Local Hoare triples

Local Hoare triple: $\langle\varphi_0\rangle s_0 \langle\psi_1\rangle$

φ_0 ψ_0

s_0: `x' := x(0)` $\oplus$ `y(0);`

φ_1 ψ_1

s_1: `y' := x` $\wedge$ `y;`

φ_2 ψ_2

s_2: `x(f) := x` $\oplus$ `y;`

φ_3 ψ_3

Local Hoare triple: $\langle\varphi_2\rangle s_2 \langle\psi_3\rangle$

A key lemma

$\langle\varphi\rangle\mathcal{P}\langle\psi\rangle$ *false* $\Leftrightarrow$ all local Hoare triples *false*.
All local Hoare triples *false* $\Leftrightarrow$ some local Hoare triple *false*.

What does this lemma mean for us?

If for some i, s_i can be fixed to make $\langle \varphi_{i-1} \rangle s_i \langle \psi_i \rangle$ *true*, then we have found $\mathcal{P}'$ such that $\langle \varphi \rangle \mathcal{P}' \langle \psi \rangle$!

This is the basis for our repair algorithm.

What does this lemma mean for us?

If for some i, s_i can be fixed to make $\langle\varphi_{i-1}\rangle s_i \langle\psi_i\rangle$ *true*, then we have found $\mathcal{P}'$ such that $\langle\varphi\rangle\mathcal{P}'\langle\psi\rangle$!

This is the basis for our repair algorithm.

Sketch of repair algorithm

- Choose promising order
- `Query` stmts in turn for repairability
  - If yes, `Repair` stmt, return modified program
  - If not, move to next stmt
- If `Query` fails for all stmts, report failure

Sketch of repair algorithm

- Choose promising order
- `Query` stmts in turn for repairability
    - If yes, `Repair` stmt, return modified program
    - If not, move to next stmt
- If `Query` fails for all stmts, report failure

Sketch of repair algorithm

- Choose promising order
- `Query` stmts in turn for repairability
  - If yes, `Repair` stmt, return modified program
  - If not, move to next stmt
- If `Query` fails for all stmts, report failure

Sketch of repair algorithm

- Choose promising order
- `Query` stmts in turn for repairability
  - If yes, `Repair` stmt, return modified program
  - If not, move to next stmt
- If `Query` fails for all stmts, report failure

Sketch of repair algorithm

- Choose promising order
- `Query` stmts in turn for repairability
  - If yes, `Repair` stmt, return modified program
  - If not, move to next stmt
- If `Query` fails for all stmts, report failure

`Query` for assignment statement

- Let $\hat{s}_i$: `vj := expr` be potential repair for s_i
- Use variable z to denote `expr` to enable formulation of Quantified Boolean Formula (QBF)

`Query` returns `yes` iff following QBF is *true* for some j:

$$\forall v_1(0)\forall v_2(0)\ldots\forall v_t(0)\exists z \ \varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,j}$$

`Query` for assignment statement

- Let $\hat{s}_i$: `v_j := expr` be potential repair for s_i
- Use variable z to denote `expr` to enable formulation of Quantified Boolean Formula (QBF)

`Query` returns `yes` iff following QBF is *true* for some j:
$$\forall v_1(0) \forall v_2(0) \ldots \forall v_t(0) \exists z \;\; \varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,j}$$

`Repair` for assignment statement

- Let m^{th} QBF be *true*
- Thus, $\hat{s}_i$: $\mathtt{v_m} := z$;
- How do we obtain z in terms of variables in $\mathcal{V}$?

$$\forall v_1(0)\forall v_2(0)\ldots\forall v_t(0)\exists z \ \underbrace{\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,m}}_{T}$$

$z = T|_z$ is a witness to QBF validity

`Repair` for assignment statement

- Let m^{th} QBF be *true*
- Thus, $\hat{s}_i$: $\mathtt{v_m}$ `:=` z`;`
- How do we obtain z in terms of variables in $\mathcal{V}$?

$$\forall v_1(0)\forall v_2(0)\ldots\forall v_t(0)\exists z \; \underbrace{\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,m}}_{T}$$

$z = T|_z$ is a witness to QBF validity

`Repair` for assignment statement

- Let m^{th} QBF be *true*
- Thus, $\hat{s}_i$: $\texttt{v}_\texttt{m}$ `:=` z;
- How do we obtain z in terms of variables in $\mathcal{V}$?

$$\forall v_1(0) \forall v_2(0) \ldots \forall v_t(0) \exists z \; \underbrace{\varphi_{i-1} \Rightarrow \widehat{\psi}_{i-1,m}}_{T}$$

$z = T|_z$ is a witness to QBF validity

Example

Pre-condition propagation

φ_0: *true*

φ_1: $x' \equiv (x(0) \oplus y(0)) \wedge$
$y' \equiv y(0)$

φ_2: $x' \equiv (x(0) \oplus y(0)) \wedge$
$y' \equiv (\neg x(0) \wedge y(0))$

φ_3: $x' \equiv (x(0) \wedge \neg y(0)) \wedge$
$y' \equiv (\neg x(0) \wedge y(0))$

Incorrect Program

```
x'  := x(0) ⊕ y(0);

y'  := x ∧ y;

x(f)  := x ⊕ y;
```

ψ_0: $y(0) \equiv (x(0) \wedge \neg y(0)) \wedge$
$x(0) \equiv (\neg x(0) \wedge y(0))$

ψ_1: $y(0) \equiv (x \wedge \neg y) \wedge$
$x(0) \equiv (x \wedge y)$

ψ_2: $y(0) \equiv x \oplus y \wedge$
$x(0) \equiv y$

ψ_3: $x(f) \equiv y(0) \wedge$
$y(f) \equiv x(0)$

Post-condition propagation

QBF for $\hat{s}_2$: $\forall x(0) \forall y(0) \exists z \; \varphi_1 \Rightarrow \widehat{\psi}_{1,y} = \mathit{true}$

Synthesized repair: `y'  := x ⊕ y;`

Complexity

Worst-case complexity exponential in # Boolean predicates

In practice, most computations are efficient using BDDs

- Symbolic storage
- Efficient manipulation of pre-/post-conditions
- Efficient computation of fix-points
- Easy QBF validity checking
- Easy cofactor computation

Complexity

Worst-case complexity exponential in # Boolean predicates

In practice, most computations are efficient using BDDs

- Symbolic storage
- Efficient manipulation of pre-/post-conditions
- Efficient computation of fix-points
- Easy QBF validity checking
- Easy cofactor computation

Extant work

- Error localization based on analyzing error traces: [Ze02], [RenRei03], [BaNaRa03], [ShQiLi04], [Gro05]
- Repair of Boolean programs: [GrBlCoo06]
- Sketching: [S-LTaBoSeSa06]
- Repair of circuits using QBFs: [StBl07]
- Dynamic repair of data structures: [DeRi03], [ElGaSuKh07]

Contributions

- Novel application of Hoare logic
- Identification of program model, fault model and specification logic for tractable repair algorithm
- Framework for repair without prior fault localization
- Exponentially lower complexity than existing algorithm ([Griesmayer et al. 2006]) for our fragment

Contributions

- Novel application of Hoare logic
- Identification of program model, fault model and specification logic for tractable repair algorithm
- Framework for repair without prior fault localization
- Exponentially lower complexity than existing algorithm ([Griesmayer et al. 2006]) for our fragment

Contributions

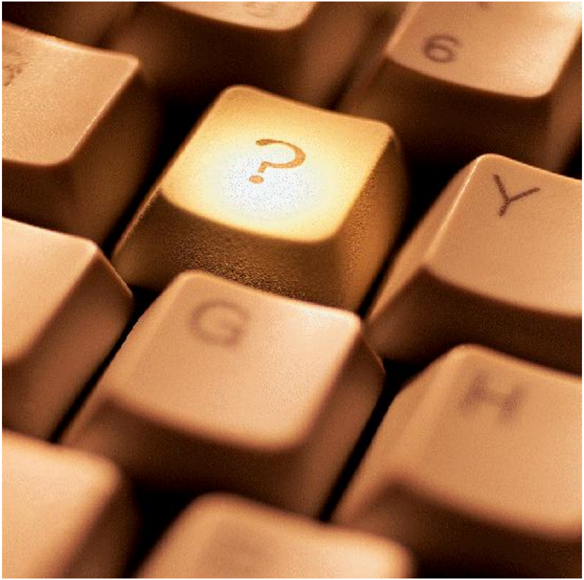
- Novel application of Hoare logic
- Identification of program model, fault model and specification logic for tractable repair algorithm
- Framework for repair without prior fault localization
- Exponentially lower complexity than existing algorithm ([Griesmayer et al. 2006]) for our fragment

Contributions

- Novel application of Hoare logic
- Identification of program model, fault model and specification logic for tractable repair algorithm
- Framework for repair without prior fault localization
- Exponentially lower complexity than existing algorithm ([Griesmayer et al. 2006]) for our fragment

The road ahead . . .

- More general fault models
  - e.g., swapped statements, multiple incorrect expressions
- Boolean programs with arbitrary recursion
- Bit-vector programs
  - VHDL or Verilog programs
  - Software programs with small integer domains


?
6
Y
G
H