School of Information Systems
SMU Singapore Management University
Carnegie Mellon University
isr institute for SOFTWARE RESEARCH

Enhancing Automated Program Repair With Deductive Verification

Xuan Bach D. Le[1], Quang-Loc Le[2], David Lo[1], Claire Le Goues[3]

[1]Singapore Management University
[2]Singapore University of Technology and Design
[3]Carnegie Mellon University

Automatic patch generation seeks to improve software quality.

- Bugs in software incur tremendous maintenance cost.

> In 2006, everyday, almost 300 bugs appear in Mozilla […] far too much for programmers to handle

- Developers presently debug and fix bugs manually.
- Automated program repair:

> APR = Fault Localization + **Repair Strategies**

Automatic patch generation seeks to improve software quality.

- Bugs in software in… …us maintenance…

In 2006, e…
[…] f…

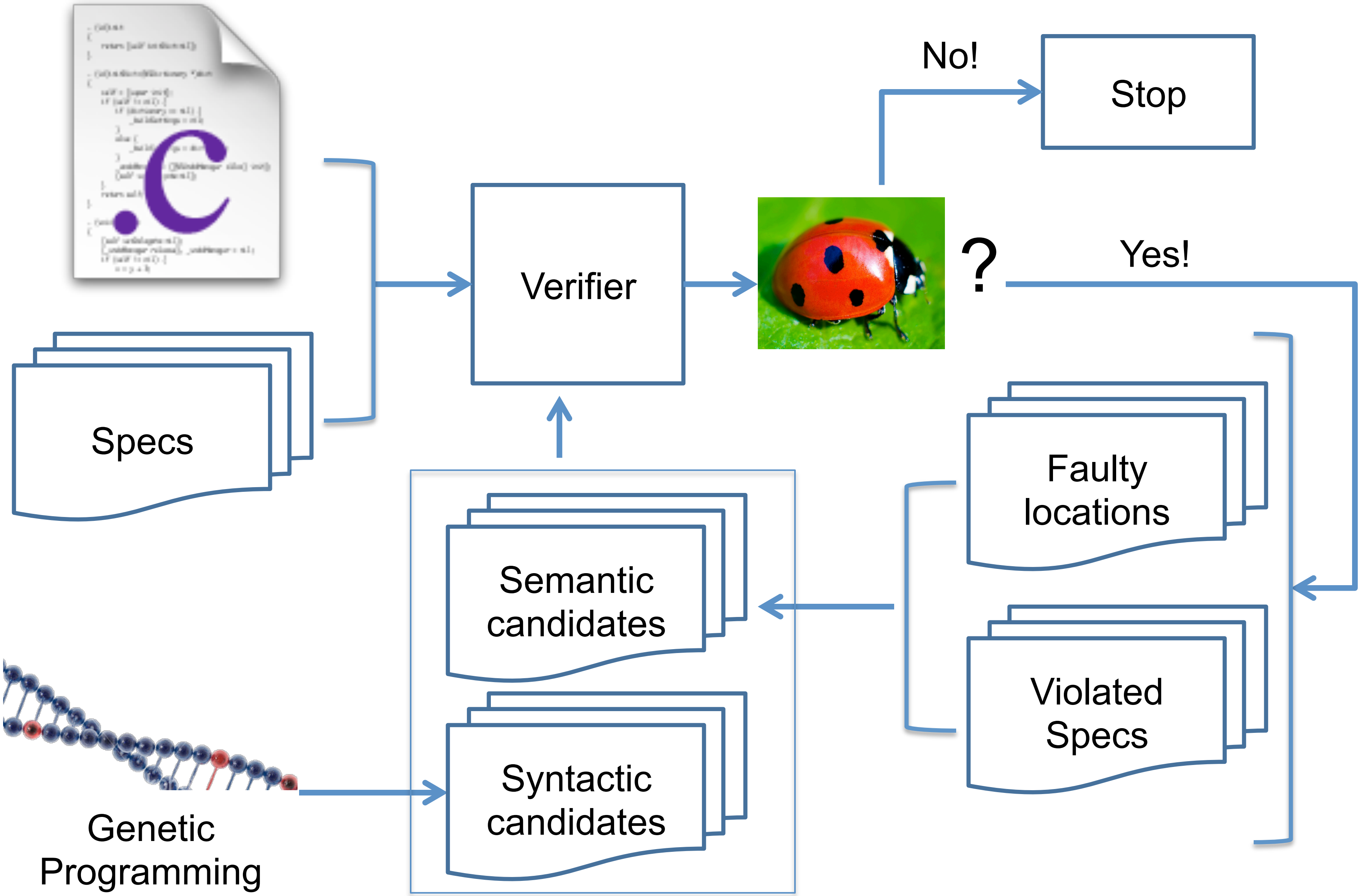
- Developers …
manually.
- Automated pr… …am repair:

APR = Fault Localization + **Repair Strategies**

1. Search: syntactic, or heuristic, “guess and check.”
2. Semantic: symbolic execution + SMT solvers, synthesis.

Benefits: more expressive than just one or the other, with correctness guarantees!

KEY IDEA: COMBINE BOTH SEARCH- AND SEMANTICS-BASED REPAIR, WITH DEDUCTIVE VERIFICATION.


No!
Stop
Verifier
?
Yes!
Specs
Faulty
locations
Semantic
candidates
Syntactic
candidates
Violated
Specs
Genetic
Programming

HIP/SLEEK: takes as input a buggy program and separation logic specification.

- Identifies components of spec that are violated.
- Localize to potentially implicated source locations/constructs:
  - Semantic: if- and loop-conditions (backwards dependency from later statements), right-hand-side of assignments.
  - Syntactic: statement level
- Verify correctness of candidate patched programs.

Example

```
bool addint (int c, int[] out, int *j, int max)







{
        bool result = false;
        if( *j >= max ) result = false;
        else{
                *j = *j + 1;
                out[*j] = c; //Bug: out array may overflow
                result = true;
        }
        return result;
}
```

Example

```
bool addint (int c, int[] out, int *j, int max)
/* @Spec req j➔ int_ref<j_val> & max >=0 & j_val <= max
case {
j_val=max -> ens j➔int_ref<j_val> & j_val’=j_val & res=false
j_val<max -> req j_val>=0 ens j➔int_ref<j_val> & j_val’=j_val+1 &
out’[j_val’-1]=c & j_val’<=max & res=true
}*/
{
        bool result = false;
        if( *j >= max ) result = false;
        else{
                *j = *j + 1;
                out[*j] = c; //Bug: out array may overflow
                result = true;
        }
        return result;
}
```

Specification language: separation Logic as supported by HIP/SLEEK

$$Y \ ::= \texttt{requires}\ \Phi\ Y \mid \texttt{case}\{\pi_1 \Rightarrow Y_1; \ldots; \pi_n \Rightarrow Y_n\} \mid \texttt{ensures}\ \Phi$$

- Example:

req j➔ int_ref<j_val> & max >=0 & j_val <= max
case {
j_val=max ->
    ens j➔int_ref<j_val> & j_val'=j_val & res=false
j_val<max ->
    req j_val>=0
    ens j➔int_ref<j_val> & j_val'=j_val+1 &
        j_val'<=max & out'[j_val'-1]=c & res=true
}

Specification language: separation Logic as supported by HIP/SLEEK

$$Y ::= \texttt{requires}\ \Phi\ Y \mid \texttt{case}\{\pi_1 \Rightarrow Y_1; \ldots; \pi_n \Rightarrow Y_n\} \mid \texttt{ensures}\ \Phi$$

- Example:

req j➔ int_ref<j_val> & max >=0 & j_val <= max
case {
j_val=max ->
ens j➔int_ref<j_val> & j_val'=j_val & res=false
j_val<max ->
req j_val>=0
ens j➔int_ref<j_val> & j_val'=j_val+1 &
j_val'<=max & **out'[j_val'-1]=c** & res=true
}

Example

```
bool addint (int c, int[] out, int *j, int max)
/* @Spec req j➔ int_ref<j_val> & max >=0 & j_val <= max
case {
j_val=max -> ens j➔int_ref<j_val> & j_val'=j_val & res=false
j_val<max -> req j_val>=0 ens j➔int_ref<j_val> & j_val'=j_val+1 &
out'[j_val'-1]=c & j_val'<=max & res=true
}*/
{
        bool result = false;
        if( *j >= max ) result = false;
        else{
                *j = *j + 1;
                out[*j] = c; //Bug: out array may overflow
                result = true;
        }
        return result;
}
```

Example

```
bool addint (int c, int[] out, int *j, int max)
/* @Spec req j➔ int_ref<j_val> & max >=0 & j_val <= max
case {
j_val=max -> ens j➔int_ref<j_val> & j_val'=j_val & res=false
j_val<max -> req j_val>=0 ens j➔int_ref<j_val> & j_val'=j_val+1 &
out'[j_val'-1]=c & j_val'<=max & res=true
}*/
{
        bool result = false;
        if( *j >= max ) result = false;
        else{
                *j = *j + 1;
                out[*j] = c; //Bug: out array may overflow
                result = true;
        }
        return result;
}
```

Semantic Candidates via Violated Specs

- Identify relevant violated sub-formula
  - Preconditions, case blocks => *expressions* of if-condition

```
case {
j_val=max -> …
j_val<max ->  …
}
```

  - Otherwise => *assignment*

out'[j_val'-1]=c → out[*j -1]=c

Syntactic Candidates via statement-level operators.

- We use genetic programming to additionally generate syntactic candidates
- Mutation operators:
  - Delete: delete a statement
  - Replace: replace a statement by another
  - Swap: swap two statements
  - Append: append a statement after another
- This helps deal with general bugs

Example

```
bool addstr (int c, int[] out, int *j, int max)
/* @Spec req j➔ int_ref<j_val> & max >=0 & j_val <= max
case {
j_val=max -> ens j➔int_ref<j_val> & j_val'=j_val & res=false
j_val<max -> req j_val>=0 ens j➔int_ref<j_val> & j_val'=j_val+1 &
out'[j_val'-1]=c & j_val'<=max & res=true
}*/
{
        bool result = false;
        if( *j >= max ) result = false;
        else{
            *j = *j + 1;
            out[*j] = c; //Bug: out array may overflow
            result = true;
        }
        return result;
}
```

Via semantic analysis

out[*j -1]=c

Syntactic candidate

Candidates Selection via Verification

- Recap: condense search space with more valuable candidates, including semantics and syntactic candidates
- Next: verify, evolve candidates, and choose best ones
  - Use static verifier for modular verification
  - Fitness function: Select candidates with fewer warnings
  - Evolve until find one passing verification

Experiments

Program	Mutated Loc	Loc	Time (minutes)	Bug Category
uniq	gline_loop	74	0.5	Incorrect
replace	addstr	855	2.8	Missing
replace	stclose	855	2.15	Missing
replace	stclose	855	2.2	Incorrect
replace	locate	855	2.5	Incorrect
replace	patsize	855	0.5	Incorrect
replace	esc	855	2.14	Incorrect
schedule3	dupp	693	0.43	Incorrect
print_tokens	ncl	1002	6.25	Missing
tcas2	IBC	302	0.15	Incorrect

Data: 10 seeded bugs from SIR benchmark
Specifications written by second author of the paper

Experiments

Program	Mutated Loc	Loc	Time (minutes)	Bug Category
uniq	gline_loop	74	0.5	Incorrect
replace	addstr	855	2.8	Missing
replace	stclose	855	2.15	Missing
replace	stclose	855	2.2	Incorrect
replace	locate	855	2.5	Incorrect
replace	patsize	855	0.5	Incorrect
replace	esc		2.14	Incorrect
schedule3	dupp		0.43	Incorrect
print_tokens	ncl	1002	6.25	Missing
tcas2	IBC	302	0.15	Incorrect

Angelix can only fix tcas2

Data: 10 seeded bugs from SIR benchmark
Specifications written by second author of the paper

Our Observations

- Angelix cannot deal with “missing implementation” bugs and is otherwise limited in the composition of its search space.
- Difference compared to our technique:
  - Angelix relies on test cases, which are an under-approximation of correctness requirements.
  - Our technique uses specs, which can express fully the desired behavior, but are less common in practice.

Conclusion

- We combine semantics-based and search-based APR via deductive verification
- We showed that:
  - Our technique fixes more bugs than state-of-the-art semantics-based APR, i.e. Angelix
  - Ensure repair soundness, mitigating overfitting.
- Future plans: automatically infer specs, experiment with different fitness functions…