Paper Review On
Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis

By Sandeep Polisetty, Xiaoyi Duan, Yuzhi Xiao

Introduction

Automated repair tools today:
GenProg, PAR, relifix,Semfix and so on.

Search-based vs sematics-based

Search base: GenProg, PAR and SPR
Semantic-base:SemFix, Nopol and Direct Fix

Important attributes for automated program repair

- Scalability
- Repairability
- Quality of repair

Balance and trade-offs between two types of repairs

SPR (search-based repair)generates more repairs and has good scalability.

Semantic-based repair has high repair quality but low scalability.

But for Angelix, it can also scale up to the same level as the most advanced search-based repair tool, using lightweight repair constraint, angelic forest and fault localization.

How does this semantic-based repair scale?

The repair constraint, angelic forest, has a size that is independent of the size of the program.

Phases of Program Repair

1. Semantic Transformation to expand class of defects repaired
2. Fault localization through heuristic based identification of possible suspicious expressions and added symbols in place
3. Constraint Generation
4. Repair generation which satisfies constraint

Example

Buggy Problem:

```
Int modulus_subtract( a,b):
    If (a>b) {
        Return  a-b
        }
    Else{
        Return a-b;
        }
```

Correct Problem

```
Int modulus_subtract( a,b):
    If (a>b) {
        Return  a-b
        }
    Else{
        Return b-a;
        }
```

Going to Step 2 Directly (Fault Localization)

Hierustic: All Statement assignments

Buggy Problem:

```
Int modulus_subtract( a,b):
        If (a>b) {
                Return  alpha;
                }
        Else{
                Return  Beta;
                }
```

Test Cases

T1: <a=1,b=2,out=1>

T2:<a=2,b=1,out=1>

Evaluate modified algorithm for each testcase

Algorithm 2 Our custom symbolic execution

Input: e^k is a k-th instance of expression e
Input: a set of suspicious expressions E
Input: σ_{sym} : $\mathit{Variables} \rightarrow \mathit{ConcreteValues} \cup \mathit{SymbolicValues}$
Output: the concrete/symbolic value of expression e^k

function EVALUATEEXPR(e^k, E, σ_{sym})
  if $e^k \in E$ **then** // if e^k is suspicious
    for $x \in$ visible variables at the location of e^k **do**
      ADDTOPATHCONDITION($x[\![e^k]\!] == \sigma_{sym}(x)$)
    end for
    // install a symbol for e^k
    return NEWSYMBOLICVARIABLE(e^k)
  else// if e^k is not suspicious
    // Evaluate e^k as usual
    return EVALUATEEXPRCONVENTIONALLY(e^k, σ_{sym})
  end if
end function

Path at the point of each symbol evaluation
Is the snapshot of all visible variables

{a: 1, b: 2} for test case 1

Constraint Equation generation

Algorithm 1 Angelic forest generation

Input: program P, test case (I, O_e)
Input: a set of suspicious expressions E
Output: angelic forest A

```
while there is an unexplored path ∧ ¬timeout do
    // perform controlled symbolic execution
    (pc, O_a) ← CONTROLLEDSYMEXE(I, E)
    R ← pc ∧ O_a = O_e
    if R is satisfiable then
        M ← GETMODEL(R) // via a constraint solver
        A ← A ∪ EXTRACTANGELICPATH(M)
    end if
end while
return A
```

=> (alpha = 1) /\ (x=1) /\ (y=2)

This can be satisfied and is appended to the angelic path of the test t1

If this constraint cant be satisfied then it is implied that modifying these symbols would not pass the testcase

Angelic Forest

Angelic Value of suspicious expression; The value of the expression that passes a particular test case

For testcase {a:1,b:2} -> alpha:1

Angelic Path: The list of all tuples <Symbol,Angelic Value. Snapshot of Visible variables> . The list of the symbol and the snapshot of all variables when a symbol is evaluated (to aid in the selection of components)

TestCase1:   {alpha,1, {a:1,b:2}},

TestCase2:   {Beta ,1,{a:2,b:1}}

Angelic Forest -> {TestCase 1:Path1,Path2 }, {TestCase 2: Path3}

Repair Generation

$$\bigvee_{\pi_i(t,E)} \bigwedge_{(e^k,v,\sigma)} \Big(\Big(\bigwedge_{x \in dom(\sigma)} x[\![e^k]\!] = \sigma(x) \Big) \wedge e^k = v \Big),$$

((x= 1) /\ (y=2)/\ (alpha =1))/\

((x= 2) /\ (y=1)/\ (Beta =1))

The generated repair must satisfy at least one path for each test case and component must be chosen to satisfy the output with the visible variable chosen as input states

Scalability??

Size of the forest is a function of number of suspicious expressions N and not of program size.

Multi line fixes can be repaired as the impact of one repair on an another is captured through the constraint equation

Can be started with a small subset of the testsuite which is later expanded.

It does not explore cases where there is no Angelic Path

Experimental Results

RQ1. Can our repair method generate repairs from large-scale real word software

RQ2. Can our repair method fix multi-location bugs?

Table 2: Experimental results

Subject	LoC	Tests	Versions	W/I Our Defect Class	Fixed Defects				Equiv. to Developer Fixes				Time (min)
					Angelix	SPR	GenProg	AE	Angelix	SPR	GenProg	AE	
wireshark	2814K	63	7	4	4	4	1	4	0	0	0	0	23
php	1046K	8471	44	12	10	18	5	7	4	8	1	2	62
gzip	491K	12	5	2	2	2	1	2	1	1	0	0	4
gmp	145K	146	2	2	2	2	1	1	2	1	0	0	14
libtiff	77K	78	24	12	10	5	3	5	3	1	0	0	14
Overall			82	32	28	31	11	19	10	11	1	2	32

Comparison with other tools

Repairability

Repair quality

Multi-location bugs

Angelix is not only scalable but also less frequently generates functionality-deleting repairs than the existing tools such as SPR and GenProg.

Only repair tool for generating (non-functionality-deleting) fixes for multi-location bugs in large-scale real-word software

Table 3: The number of defects exclusively repaired by each repair tool across the subjects

Subject	Angelix	SPR	GenProg	AE
wireshark	0	0	0	0
php	0	4	0	0
gzip	1	0	0	0
gmp	0	0	0	0
libtiff	5	0	0	0
Overall	6	4	0	0

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
```

(a) The buggy location of libtiff-d13be72c-ccadf48a

```
if (td->td_nstrips > 2
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1])
```

(b) The repair generated by our tool, Angelix

```
if (td->td_nstrips > 1
    && td->td_compression == COMPRESSION_NONE
    && td->td_stripbytecount[0] != td->td_stripbytecount[1]
    && !(1))
```

(c) The repair generated by SPR

Table 4: The number of functionality-deleting repairs

Subject	Angelix			SPR		
	Fixes	Del	Per	Fixes	Del	Per
wireshark	4	1	25%	4	1	25%
php	10	3	30%	18	7	39%
gzip	2	0	0%	2	1	50%
gmp	2	0	0%	2	0	0%
libtiff	10	2	20%	5	4	80%
Overall	28	6	21%	31	13	42%

Table 5: Experimental results for multi-location defects.

Defect	Fixed Expressions
libtiff-4a24508-cc79c2b	2
libtiff-829d8c4-036d7bb	2
coreutils-00743a1f-ec48bead	3
coreutils-1dd8a331-d461bfd2	2
coreutils-c5ccf29b-a04ddb8d	3

Heartbleed Bug

```
if (hbtype == TLS1_HB_REQUEST) {
    ...
    memcpy (bp, pl, payload);
    ...
}
```

(a) The buggy part of the Heartbleed-vulnerable OpenSSL

```
if (hbtype == TLS1_HB_REQUEST
    && payload + 18 < s->s3->rrec.length) {
    /* receiver side: replies with TLS1_HB_RESPONSE */
}
```

(b) A fix generated by our tool, Angelix

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0;
...
if (hbtype == TLS1_HB_REQUEST) {
    /* receiver side: replies with TLS1_HB_RESPONSE */
}
else if (hbtype == TLS1_HB_RESPONSE) {
    /* sender side */
}
return 0;
```

(c) The developer-provided repair

Figure 4: The Heartbleed bug and their fixes

Threats to Validity

1) Subject programs in the existing benchmark previously used to evaluate GenProg, AE and SPR. The validity of the experimental results are limited.

2) Configurations (the maximum number of suspicious locations)

3) Components particular to the chosen defect class

Conclusion

A semantic-based repair method

Novel lightweight repair repair constraint called ‘angelic forest’

Better repair quality

Successfully fixed multi-location bugs

Questions

1. By using the MaxSMT how can it guarantee minimal change, as the solver by definition would try to satisfy as many components as possible ? This should increase the size of the change

2. Would this work on composite components ?

3. How is loop unrolling done for For loops ?

4. Would the search space increase exponentially as the number of inputs to components increase ?

5. When solving the constraint, can expressions that are satisfied by existing code removed from the constraint equation ?

Thank you!