The Modification Challenge

Program $P$

Verifier

Property $\varphi$

✓

✗

Challenges

- Programs change frequently
- Reverification after each change required
- Reverification must keep up with change rate

⇒ Verifying modified programs from scratch infeasible
The Modification Challenge

Program $P \rightarrow \text{modifications} \rightarrow \text{Program } P'$

- bug fix
- refactoring
- change request
- new feature

Verifier

Property $\varphi$

$P' \models \varphi$?

new verification task
The Modification Challenge

Program $P \rightarrow P'$

- modification
  - bug fix
  - refactoring
  - change request
  - new feature

Verifier

Property $\varphi$

$P' \models \varphi$?

new verification task

**Challenge**

- Programs change frequently
- Reverification after each change required
- Reverification must keep up with change rate

$$\implies$$ Verifying modified programs from scratch infeasible
Insights about Program Modification

**Observation:** Program modifications typically affect few program executions
Insights about Program Modification

**Observation:** Program modifications typically affect few program executions

Original program $P$

```
0 r=0;
1 if(a<0)  
2   while(a<0)  
3     r=r-a;
4   a=a+1;
   else
5   r=a+a+1;
6   r=r/2;
7 assert r>0;
```

Modified program $P'$

```
0 r=0;
1 if(a<0)  
2   while(a<0)  
3     r=r-a;
4   a=a+1;
   else
5   r=a*(a+1);
6   r=r/2;
7 assert r>0;
```
Insights about Program Modification

**Observation**: Program modifications typically affect few program executions

Original program $P$

```
0 r=0;
1 if(a<0)
2  while(a<0)
3     r=r-a;
4   a=a+1;
   else
5   r=a+a+1;
6   r=r/2;
7 assert r>0;
```

Modified program $P'$

```
0 r=0;
1 if(a<0)
2  while(a<0)
3     r=r-a;
4   a=a+1;
   else
5   r=a*(a+1);
6   r=r/2;
7 assert r>0;
```
**Insights about Program Modification**

**Observation:** Program modifications typically affect few program executions

Original program $P$

1. $r=0$
2. if($a<0$)
3. while($a<0$)
4. $r=r-a$
5. $a=a+1$
6. else
7. $r=r/2$
8. assert $r>0$

Modified program $P'$

1. $r=0$
2. if($a<0$)
3. while($a<0$)
4. $r=r-a$
5. $a=a+1$
6. else
7. $r=a*(a+1)$
8. $r=r/2$
9. assert $r>0$

$\Rightarrow$ verifying unchanged program executions from scratch wastes resources

$\Rightarrow$ **better** reuse information obtained in previous verification run(s)
Principle of Incremental Verification

**General idea:** Use information about previous verification run

⇒ speed up verification of unchanged parts
Weaknesses of Existing Approaches

- Program $P \rightarrow \text{Incremental verifier X}
- \text{used facts, etc.} \rightarrow \times
- \text{initial facts, etc.} \rightarrow \checkmark
- Modified program $P' \rightarrow \text{Incremental verifier X}
- \times \rightarrow \times

Often,
- Require initial full verification
- Use same verifier in each step
- Tailored to specific verification approach
Weaknesses of Existing Approaches

- Often:
  - Require initial full verification
  - Use same verifier in each step
  - Tailored to specific verification approach

- Equivalence checking:
  - Does not verify property
  - Modified program only as correct as original
  - Can be expensive

```
Program P  -->  Incremental verifier X
                 ↑
                 ✓
used facts, etc.  ×

Modified program P'  -->  Incremental verifier X
                 ↑
                 ✓
initial facts, etc.  ×
```
**Differential Modular Software Verification**

**The Basic Idea**

- **Differential Condition Generator**
- **Conditional Verifier**
- **Verifier**

- **Reduced-based conditional verifier**

**Soundness Criterion**
Generated condition does not cover any modified execution, i.e., it does not cover executions from $ex(P') \setminus ex(P)$. 
Properties of Modular Differential Software Verification

- Unchanged program parts are as correct as before
- Only verify modified program parts
- Applicable to arbitrary verifiers due to reducer-based conditional model checking
- Verifier can be newly selected for each modified program
1. Detect differences
   - Compute the parallel composition of original and modified program
   - Stop if modified program deviates
## Syntactic Differential Condition Generator

1. **Detect differences**
   - Compute the parallel composition of original and modified program
   - Stop if modified program deviates

<table>
<thead>
<tr>
<th>Original program $P$</th>
<th>Modified program $P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 $r=0;$</td>
<td>0 $r=0;$</td>
</tr>
<tr>
<td>1 if ($a&lt;0$)</td>
<td>1 if ($a&lt;0$)</td>
</tr>
<tr>
<td>2 while ($a&lt;0$)</td>
<td>2 while ($a&lt;0$)</td>
</tr>
<tr>
<td>3 $r=r-a;$</td>
<td>3 $r=r-a;$</td>
</tr>
<tr>
<td>4 $a=a+1;$</td>
<td>4 $a=a+1;$</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>5 $r=a+a+1;$</td>
<td>5 $r=a*(a+1);$</td>
</tr>
<tr>
<td>6 $r=r/2;$</td>
<td>6 $r=r/2;$</td>
</tr>
<tr>
<td>7 assert $r&gt;0;$</td>
<td>7 assert $r&gt;0;$</td>
</tr>
</tbody>
</table>
1. Detect differences
- Compute the parallel composition of original and modified program
- Stop if modified program deviates

Original program $P$

```
0 r=0;
1 if(a<0)
2   while(a<0)
3     r=r-a;
4     a=a+1;
5     !a<0;  // Assert r > 0;
6     r=r/2;
7 assert r>0;
```

Modified program $P'$

```
0 r=0;
1 if(a<0)
2   while(a<0)
3     r=r-a;
4     a=a+1;
5     !a<0;  // Assert r > 0;
6     r=r/2;
7 assert r>0;
```

Diagram:

- $(l_0, l'_0)$
- $(l_1, l'_1)$
- $(l_2, l'_2)$
- $(l_3, l'_3)$
- $(l_4, l'_4)$
- $(l_5, l'_5)$
- $(l_7, l'_7)$
- $(l_8, l'_8)$

Theorem (proven): Syntactical generator fulfills soundness criterion
1. Detect differences
   - Compute the parallel composition of original and modified program
   - Stop if modified program deviates

Original program $P$

0 $r=0$;
1 if ($a < 0$)
2 while ($a < 0$)
3 $r = r - a$;
4 $a = a + 1$;
else
5 $r = a + a + 1$;
6 $r = r / 2$;
7 assert $r > 0$;

Modified program $P'$

0 $r = 0$;
1 if ($a < 0$)
2 while ($a < 0$)
3 $r = r - a$;
4 $a = a + 1$;
else
5 $r = a * (a + 1)$;
6 $r = r / 2$;
7 assert $r > 0$;
1. Detect differences
   - Compute the parallel composition of original and modified program
   - Stop if modified program deviates

<table>
<thead>
<tr>
<th>Original program $P$</th>
<th>Modified program $P'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0 \ r=0;$</td>
<td>$0 \ r=0;$</td>
</tr>
<tr>
<td>$1 \ if(a&lt;0)$</td>
<td>$1 \ if(a&lt;0)$</td>
</tr>
<tr>
<td>$2 \ while(a&lt;0)$</td>
<td>$2 \ while(a&lt;0)$</td>
</tr>
<tr>
<td>$3 \ r=r-a;$</td>
<td>$3 \ r=r-a;$</td>
</tr>
<tr>
<td>$4 \ a=a+1;$</td>
<td>$4 \ a=a+1;$</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>$5 \ r=a+a+1;$</td>
<td>$5 \ r=a*(a+1);$</td>
</tr>
<tr>
<td>$6 \ r=r/2;$</td>
<td>$6 \ r=r/2;$</td>
</tr>
<tr>
<td>$7 \ assert r&gt;0;$</td>
<td>$7 \ assert r&gt;0;$</td>
</tr>
</tbody>
</table>

2. Generate condition from ARG of parallel composition (standard procedure)
1. Detect differences
   - Compute the parallel composition of original and modified program
   - Stop if modified program deviates

Original program $P$

```
0 r=0;
1 if(a<0) goto 2;
2 while(a<0) r=r-a;
3 r=a+1;
4 r=a+a+1;
5 r=r/2;
6 assert r>0;
```

Modified program $P'$

```
0 r=0;
1 if(a<0) goto 2;
2 while(a<0) r=r-a;
3 r=a+1;
4 else r=a*(a+1);
5 r=r/2;
6 assert r>0;
```

2. Generate condition from ARG of parallel composition (standard procedure)

**Theorem** (proven): Syntactical generator fulfills soundness criterion
Evaluation Setting

Environment

• Similar to SV-COMP
• But only granted 4 cores
Evaluation Setting

Environment
- Similar to SV-COMP
- But only granted 4 cores

Verifiers
- Predicate-based, native conditional model checker
- CPASeq
- Ultimate Automizer
Evaluation Tasks

- Combinations of two SV-COMP tasks

```c
void main() {
    if(__VERIFIER_nondet_int())
        main1();
    else
        main2();
}
```
Evaluation Tasks

- Combinations of two SV-COMP tasks
  ```c
  void main() {
    if(__VERIFIER_nondet_int())
      main1();
    else
      main2();
  }
  ```
- (1) eca+token, (2) gcd+newton, (3) pals+eca, (4) sfifo+token, (5) square+soft, and (6) ssh (client+server)
Evaluation Tasks

• Combinations of two SV-COMP tasks
  
  ```
  void main() {
    if(__VERIFIER_nondet_int())
      main1();
    else
      main2();
  }
  ```

• (1) eca+token, (2) gcd+newton, (3) pals+eca, (4) sfifo+token, (5) square+soft, and (6) ssh (client+server)

• Program Modification
  
  • Replace one SV-COMP task by another of same category
  • Either fix an unsafe program or
  • Introduce a bug in a safe program
    
    Full verification and our approach find the same violations
Typically, modular differential software verification similar or better
• Modular differential software verification can be better
• Full verification often better
  (modular differential software verification suffers from residual programs)
Comparison with Precision Reuse

- Modular differential software verification often better than precision reuse
- Combination of two approaches often not beneficial
Conclusion

Proposed differential modular software verification

Identified weaknesses
- Evaluation only on artificial benchmarks
- Syntactic difference often too imprecise in practice
- Suffers from residual program structure