From Underapproximations to Overapproximations and Back!

Arie Gurfinkel
Software Engineering Institute
Carnegie Mellon University

joint work with Aws Albarghouthi and Marsha Chechik from University of Toronto
NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

Use of any trademarks in this presentation is not intended in any way to infringe on the rights of the trademark holder.

This Presentation may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.
Software Engineering Institute (SEI)

Department of Defense R&D Laboratory (FFRDC)
Created in 1984
Under contract to Carnegie Mellon University
Offices in Pittsburgh, PA; Washington, DC; and Frankfurt, Germany

SEI Mission: advance software engineering and related disciplines to ensure the development and operation of systems with predictable and improved cost, schedule, and quality.
Software is Everywhere
Software is Full of Bugs!

“Software easily rates as the most poorly constructed, unreliable, and least maintainable technological artifacts invented by man”

Paul Strassman, former CIO of Xerox
Software Bugs are Expensive!

Intel Pentium FDIV Bug
- Estimated cost: $500 Million

Y2K bug
- Estimated cost: >$500 Billion

Northeast Blackout of 2003
- “a programming error identified as the cause of alarm failure”
- Estimated cost: $6-$10 Billion

“The cost of software bugs to the U.S. economy is estimated at $60 B/year”
NIST, 2002
Some Examples of Software Disasters

Between 1985 and 1987, Therac-25 gave patients massive overdoses of radiation, approximately 100 times the intended dose. Three patients died as a direct consequence.

On February 25, 1991, during the Gulf War, an American Patriot Missile battery in Dharan, Saudi Arabia, failed to track and intercept an incoming Iraqi Scud missile. The Scud struck an American Army barracks, killing 28 soldiers and injuring around 100 other people.

On June 4, 1996 an unmanned Ariane 5 rocket launched by the European Space Agency forty seconds after lift-off. The rocket was on its first voyage, after a decade of development costing $7 billion. The destroyed rocket and its cargo were valued at $500 million.

Details at http://www5.in.tum.de/~huckle/bugse.html
Recent Examples

In July 2010, The Food and Drug Administration ordered Baxter International to recall all of its Colleague infusion pumps in use and provide a refund or no-cost replacement to United States customers. It has been working with Baxter since 1999 to correct numerous device flaws. Some of the issues were caused by simple buffer overflow.

In December 2010, the Skype network went down for 3 days. The source of the outage was traced to a software bug in Skype version 5.

In January 2011, two German researchers have shown that most “feature” mobile phones can be “killed” by sending a simple SMS message (**SMS of Death**). The attack exploits many bugs in the implementation of SMS protocol in the phones. It can potentially bring down all mobile communication…
Why so many bugs?

Software Engineering is very complex

• Complicated algorithms
• Many interconnected components
• Legacy systems
• Huge programming APIs
• ...

Software Engineers need better tools to deal with this complexity!
What Software Engineers Need Are …

Tools that give better confidence than testing while remaining easy to use

And at the same time, are

• … fully automatic
• … (reasonably) easy to use
• … provide (measurable) guarantees
• … come with guidelines and methodologies to apply effectively
• … apply to real software systems
Automated Software Analysis

Program  →  Automated Analysis  ←  Correct  Incorrect

Software Model Checking with Predicate Abstraction
  e.g., Microsoft’s SDV

Abstract Interpretation with Numeric Abstraction
  e.g., ASTREE, Polyspace
Outline of The Rest

Over- and Under-approximation Driven Approaches

UFO: From Under- to Over- and Back!

Exploration Strategy

Refinement Strategy

Conclusion
Overapproximation-driven Approach (CEGAR)

Program → Compute invariant using abstract post → Is safe?

[Flowchart Diagram]

Interpolation or WP

e.g., BLAST, SLAM, CPAChecker, YaSM, SATAbs, etc.
Is ERROR Reachable?

Program

Abstraction

Over-approximation

1: int x = 2;
2: while (y <= 2)
3:     y = y - 1;
4: if (x == 2)
5:     ERROR;
6: ;

1: 
2: while (*)
3:     
4: if (*)
5:     ERROR;
6: 

CEGAR steps

Abstract  Translate  Check  Validate
**Over-Driven: Is ERROR Reachable?**

### Program

1: int x = 2;
2: int y = 2;
3: while (y <= 2)
4:   y = y - 1;
5: if (x == 2)
6:   ERROR;

### Abstraction (with y<=2)

bool b is (y <= 2)
1: b = T;
2: while (b)
3:   b = b ? T : *;
4: if (*)
5:   ERROR;
6:  

### Over-Approximation

1: 
2: b=F
3: b=T
4: b=F
5: b=F
6: b=F

UNREACHABLE

### CEGAR steps

Abstract  ➔ Translate  ➔ Check  ➔ NO ERROR
Underapproximation-driven Approach (Impact)

Program

Generate paths to error

Are these paths feasible?

Cex

No

SMT

No

Interpolation/ WP

Safe

Is result an inductive invariant?

Explain why safe

e.g., Impact, Impact2, Synergy, Dash, Wolverine
Under-Driven: Is ERROR Reachable?

Program

1: int x = 2;
2: int y = 2;
3: while (y <= 2)
4:   y = y - 1;
5: if (x == 2)
6:   ERROR;;

IMPACT steps

Explore → Refine → Explore → Refine → Cover
Over-Driven v.s. Under-Driven in a Nutshell

```c
int main()
{
    1 ...
    2 while (...){
        ...
    }
    E: ERROR
}
```
Over-Driven v.s. Under-Driven in a Nutshell

```
int main() {
  1 ...
  2 while (...){
    ...
  }
E: ERROR
}
```

Explore
Refine
Explore
Over-Driven v.s. Under-Driven in a Nutshell

```c
int main(){
  1 ...
  2 while (...){
    ...
  }
  E: ERROR
}
```

Explore
Refine
Explore

Explore
Refine
Explore

Unlabeled
Pred. abs. label
Interpolant label
OD vs. UD Approaches

OD

UD

Number of Refinements

Cost of Exploration
Our Algorithm: UFO

UD algorithm

Interpolation-based

Combination of UD and OD

OD algorithm

Predicate abstraction based

A novel interpolation-based refinement
Multiple paths checked and refined with a single SMT call
UFO in a Nutshell

Iteration 1

Imprecise post → UD
Explore from root → OD

Iteration 2

Refinement

Unlabeled
Pred. abs. label
Interpolant label
The UFO Algorithm

1: func UfoMain (Program P) :
2:    create node \( v_{en} \)
3:    \( \psi(v_{en}) \leftarrow true, \nu(v_{en}) \leftarrow T \)
4:    marked\((v_{en}) \leftarrow true
5:    labels \leftarrow \emptyset
6:    while true do
7:        EXPANDARG()
8:        if \( \psi(v_{en}) \) is UNSAT then
9:            return SAFE
10:       labels \leftarrow Refine()
11:       if labels = \emptyset then
12:           return UNSAFE
13:       clear AH and FN
14:    end while
15:    if FN(\ell) exists then
16:        return FN(\ell)
17:    end if
18:    create node \( v \)
19:    \( \psi(v) \leftarrow true, \nu(v) \leftarrow \ell \)
20:    FN(\ell) \leftarrow v
21:    return v

22: func EXPANDNODE \((v \in V)\) :
23:    if \( v \) has children then
24:        for all \((v, w) \in E\) do
25:            FN(\nu(w)) \leftarrow w
26:    else
27:        for all \((\nu(v), T, \ell) \in \Delta\) do
28:            w \leftarrow GETFUTURENODE(\ell)
29:            E \leftarrow E \cup \{(v, w)\}; \tau(v, w) \leftarrow T
30:    end if
31:    while true do
32:        EXPANDNODE(\(v\))
33:        if marked(\(v\)) then
34:            marked(\(v\)) \leftarrow false
35:            \( \psi(v) \leftarrow \bigvee_{(u, v) \in E} Post(u, v) \)
36:            for all \((v, w) \in E\) do marked\((w) \leftarrow true
37:        else if labels(\(v\)) bound then
38:            \( \psi(v) \leftarrow labels(\(v\)) \)
39:            for all \(\{(v, w) \in E \mid labels(\(w\)) unbound\}\) do
40:                marked\((w) \leftarrow true
41:        if \( v = v_{en} \) then break
42:        if \( \nu(\(v\)) \) is head of a component then
43:            if \( \psi(v) \Rightarrow \bigvee_{u \in AH(\nu(v))} \psi(u) \) then
44:                erase AH(\(\nu(v)\)) and FN(\(\nu(v)\))
45:                \( l \leftarrow WtoExit(\nu(\(v\))) \)
46:                \( v \leftarrow FN(\ell); \) erase FN(\(\ell\))
47:                for all \(\{(v, w) \in E \mid \not\exists u \neq v \cdot (u, w) \in E\}\) do
48:                    erase FN(\(w\))
49:                continue
50:                add \( v \) to AH(\(\nu(v)\))
51:                \( l \leftarrow WtoNext(\nu(\(v\))) \)
52:                \( v \leftarrow FN(\ell); \) erase FN(\(\ell\))
Weak Topological Ordering

Definition (WTO):
A weak topological order (WTO) of a DAG $G = (V, E)$ is a well-parenthesised total-order $\preceq$ of $V$ without two consecutive ‘(‘ such that for every edge $(u, v) \in E$:

$$(u < v \land v \notin \omega(u)) \lor (u \preceq u \land v \in \omega(u))$$

Elements between two matching paren. are called components

First element of a component is called head

$\omega(u)$ is the set of heads of components containing $u$
Refinement
Craig Interpolation Theorem

**Theorem** (Craig 1957)
Let $A$ and $B$ be two First Order (FO) formulae such that $A \Rightarrow \neg B$, then there exists a FO formula $I$, denoted $ITP(A, B)$, such that

\[
A \Rightarrow I \quad I \Rightarrow \neg B \quad \text{atoms}(I) \in \text{atoms}(A) \cap \text{atoms}(B)
\]

**Theorem** (McMillan 2003)
A Craig interpolant $ITP(A, B)$ can be effectively constructed from a resolution proof of unsatisfiability of $A \land B$

In Model Checking, Craig Interpolation Theorem is used to safely over-approximate the set of (finitely) reachable states
Craig Interpolation in Model Checking

Over-Approximating Reachable States

- Let $R^i$ be the $i$th step of a transition system
- Let $A = \text{Init} \land R^0 \land \ldots \land R^n$ and $B = \text{Bad}$
- ITP ($A$, $B$) (if exists) is an over-approx of states reachable in $n$-steps that does not contain any Bad states
Interpolation Sequence

Given a sequence of formulas $A = \{A_i\}_{i=0}^n$, an interpolation sequence $\text{ItpSeq}(A) = \{I_1, \ldots, I_{n-1}\}$ is a sequence of formulas such that

- $I_k$ is an ITP $(A_0 \land \ldots \land A_{k-1}, A_k \land \ldots \land A_n)$, and
- $\forall k < n. I_k \land A_{k+1} \Rightarrow I_{k+1}$

If $A_i$ is a transition relation of step $i$, then the interpolation sequence is a proof why a program trace is safe.
DAG Interpolants: Solving the Refinement Prob.

Given a DAG $G = (V, E)$ and a labeling of edges $\pi: E \rightarrow \text{Expr}$. A **DAG Interpolant** (if it exists) is a labeling $I: V \rightarrow \text{Expr}$ such that

- for any path $v_0, \ldots, v_n$, and $0 < k < n$, $I(v_k) = ITP (\pi(v_0) \land \ldots \land \pi(v_{k-1}), \pi(v_k) \land \ldots \land \pi(v_n))$
- $\forall (u, v) \in E. (I(u) \land \pi(u, v)) \Rightarrow I(v)$

$\begin{align*}
I_2 &= ITP (\pi_1, \pi_8) \\
I_2 &= ITP (\pi_1, \pi_2 \land \pi_3 \land \pi_6 \land \pi_7) \\
\ldots
\end{align*}$

$\begin{align*}
(I_1 \land \pi_1) &\Rightarrow I_2 \\
(I_2 \land \pi_8) &\Rightarrow I_7 \\
(I_2 \land \pi_2) &\Rightarrow I_3 \\
\ldots
\end{align*}$
DAG Interpolation Algorithm

Reduce DAG Interpolation to Sequence Interpolation!

\[
\text{DagItp} ((V, E), \pi) \\
\{ \\
(A_0, \ldots, A_n) = \text{Encode}(V, E, \pi) \\
(I_1, \ldots, I_{n-1}) = \text{SeqItp}(A_0, \ldots, A_n) \\
\text{for } i \text{ in } [1, n-1] \text{ do } J_i = \text{Clean}(I_i) \\
\text{return } (J_1, \ldots, J_{n-1}) \\
\}
\]

Encode input DAG by a set of constraints. One constraint per vertex.

Compute interpolant sequence. One interpolant per vertex.

Remove out-of-scope variables
DagItp: Encode

\[ A_1 \quad v_1 \Rightarrow v_2 \land \pi_1 \]

\[ A_2 \quad v_2 \Rightarrow (v_3 \land \pi_2) \lor (v_7 \land \pi_8) \]

\[ A_3 \quad v_3 \Rightarrow (v_4 \land \pi_3) \lor (v_5 \land \pi_4) \]

\[ A_4 \quad v_4 \Rightarrow v_6 \land \pi_6 \]

\[ A_5 \quad v_5 \Rightarrow v_6 \land \pi_5 \]

\[ A_6 \quad v_6 \Rightarrow v_7 \land \pi_7 \]
DagItp: Sequence Interpolate

\[ A_1 \quad v_1 \Rightarrow v_2 \land \pi_1 \]

\[ A_2 \quad v_2 \Rightarrow (v_3 \land \pi_2) \lor (v_7 \land \pi_8) \]

\[ A_3 \quad v_3 \Rightarrow (v_4 \land \pi_3) \lor (v_5 \land \pi_4) \]

\[ A_4 \quad v_4 \Rightarrow v_6 \land \pi_6 \]

\[ A_5 \quad v_5 \Rightarrow v_6 \land \pi_5 \]

\[ A_6 \quad v_6 \Rightarrow v_7 \land \pi_7 \]
DagItp: Clean

\[ \text{Clean}(I_i) = \forall \{ x \mid x \in \text{var}(I_i) \land \neg \text{inScope}(x, v_i) \} \cdot \forall \{ v_j \mid v_j \in V \} \cdot I[v_i \leftarrow \top] \]
**UFO Refinement**

1. Construct DAG of current unfolding
2. Use DagItp to find new labels

Refinement is done with a **single** SMT call

Cleaning the labels with quantifier elimination is a major bottleneck
UFO in a Nutshell

Iteration 1

Imprecise post $\rightarrow$ UD
Explore from root $\rightarrow$ OD

Iteration 2

Refinement

Unlabeled
Pred. abs. label
Interpolant label
UFO Framework: Architecture

Preprocessing Phase

Program $P$ with assertions → C to LLVM → bitcode → Optimizations for verification → Optimized program $P^{o}$ → Cutpoint Graph (CG) constructor → CG → Weak Topological Ordering (WTO)

Analysis Phase

Interpolating SMT solver → SMT solver interface → ARG Constructor (main algorithm) → Refiner → Expansion strategy → Abstract post → Counterexample or certificate of correctness
Implementation

Implemented 5 instances of UFO

UD
- **ufoNo**: pure interpolation-based

Combined UD+OD
- **ufoCP**: interpolation with Cartesian abstraction
- **ufoBP**: interpolation with Boolean abstraction

OD
- **CP**: Cartesian predicate abstraction
- **BP**: Boolean predicate abstraction
Evaluation

Benchmarks from SV-COMP 2012:
• ntdrivers-simplified, ssh-simplified, and systemc

Pacemaker benchmarks from [VMCAI 2012]

Total 105 C programs

Compared with Wolverine
• a freely available implementation of IMPACT algorithm
• based on CProver framework
• bit-precise (our implementation is not)
## Results: Summary

<table>
<thead>
<tr>
<th></th>
<th>#SOLVED</th>
<th>#SAFE</th>
<th>#UNSAFE</th>
<th>TOT. TIME (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ufoNo</td>
<td>78</td>
<td>22</td>
<td>56</td>
<td>8,289</td>
</tr>
<tr>
<td>ufoCP</td>
<td>79</td>
<td>22</td>
<td>57</td>
<td>7,838</td>
</tr>
<tr>
<td>ufoBP</td>
<td>69</td>
<td>17</td>
<td>52</td>
<td>11,260</td>
</tr>
<tr>
<td>CP</td>
<td>49</td>
<td>10</td>
<td>39</td>
<td>15,363</td>
</tr>
<tr>
<td>BP</td>
<td>71</td>
<td>19</td>
<td>52</td>
<td>10,018</td>
</tr>
<tr>
<td>Wolverine</td>
<td>38</td>
<td>18</td>
<td>20</td>
<td>19,753</td>
</tr>
</tbody>
</table>
## Results: A Closer Look (SAFE)

<table>
<thead>
<tr>
<th></th>
<th>ufoNo</th>
<th>ufoCP</th>
<th>ufoBP</th>
<th>BP</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>#REF</td>
<td>TIME</td>
<td>#REF</td>
<td>TIME</td>
</tr>
<tr>
<td>token1</td>
<td>98</td>
<td>18</td>
<td>24</td>
<td>10</td>
</tr>
<tr>
<td>token2</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>token3</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>token4</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>token5</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>srvr1a</td>
<td>5.2</td>
<td>10</td>
<td>5.16</td>
<td>8</td>
</tr>
<tr>
<td>srvr1b</td>
<td>1.37</td>
<td>7</td>
<td>2.9</td>
<td>7</td>
</tr>
<tr>
<td>srvr2</td>
<td>171</td>
<td>17</td>
<td>184</td>
<td>17</td>
</tr>
<tr>
<td>srvr3</td>
<td>133</td>
<td>17</td>
<td>147</td>
<td>17</td>
</tr>
<tr>
<td>srvr4</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>srvr8</td>
<td>101</td>
<td>14</td>
<td>115</td>
<td>14</td>
</tr>
</tbody>
</table>
## Results: A Closer Look (UNSAFE)

<table>
<thead>
<tr>
<th></th>
<th>ufoNo</th>
<th>ufoCP</th>
<th>ufoBP</th>
<th>BP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TIME</td>
<td>#REF</td>
<td>TIME</td>
<td>#REF</td>
</tr>
<tr>
<td>kundu1</td>
<td>--</td>
<td>--</td>
<td>24</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>kundu2</td>
<td>1.24</td>
<td>2</td>
<td>2.74</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toy1</td>
<td>96</td>
<td>10</td>
<td>79</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>toy2</td>
<td>12</td>
<td>5</td>
<td>60</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>token12</td>
<td>27</td>
<td>4</td>
<td>14</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>token13</td>
<td>37</td>
<td>4</td>
<td>34</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>token14</td>
<td>10</td>
<td>3</td>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>token15</td>
<td>52</td>
<td>4</td>
<td>34</td>
<td>4</td>
</tr>
</tbody>
</table>
Results: Observations

UFO is very competitive on SV-COMP benchmarks

UFO outperforms Lazy Abstraction with Interpolants
  • i.e., Wolverine

Different instantiations are more suited to different problems

ufoCP hits the sweet spot (most consistent)

Need to experiment with different abstract domains and strategies
## Recent Related Work

### Intra-procedural

**Impact [McMillan 06]**
- Original lazy abstraction with interpolants

**Impact2 [McMillan 10]**
- Targets testing/exploration

**Wolverine [Weissenbacher 11]**
- Bit-level interpolants

**Ultimate [Ermis et al. 12]**
- Impact with Large Block Encoding for Refinement

### Inter-procedural

**Whale [Our work 12]**
- Inter-procedural verification with interpolants

**FunFrog [Sery et al. 11]**
- Function summarization using interpolants
Conclusion

UFO

• A Combined UD+OD technique
• DAG interpolation-based refinement procedure
• Extensive Evaluation on SV-COMP benchmarks
  – Results show synergy between UD and OD

Current and Future Work

• Open Source release of the UFO framework
• UFO as a verification framework [CAV 2012]
• UFO as refinement of abstract interpretations [SAS 2012]
• Inter-procedural extension of UFO via [VMCAI 2012]
Thank You!

http://www.cs.toronto.edu/~aws/ufo
Contact Information

Presenter
Arie Gurfinkel
RTSS
Telephone: +1 412-268-7788
Email: arie@cmu.edu

Web:
www.sei.cmu.edu
http://www.sei.cmu.edu/contact.cfm

U.S. mail:
Software Engineering Institute
Customer Relations
4500 Fifth Avenue
Pittsburgh, PA 15213-2612
USA

Customer Relations
Email: info@sei.cmu.edu
Telephone: +1 412-268-5800
SEI Phone: +1 412-268-5800
SEI Fax: +1 412-268-6257
THE END