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

Software Engineering Institute (SEI)

Department of Defense R&D Laboratory (FFRDC)

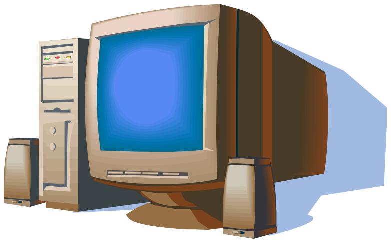
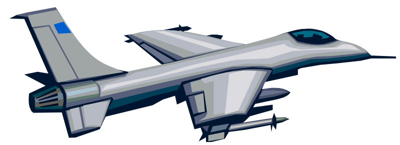
Created in 1984

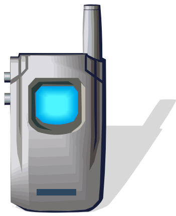
Under contract to Carnegie Mellon University

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.

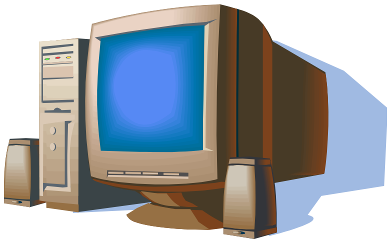
Software is Everywhere

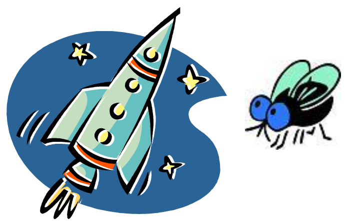








Software is Full of Bugs!





"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

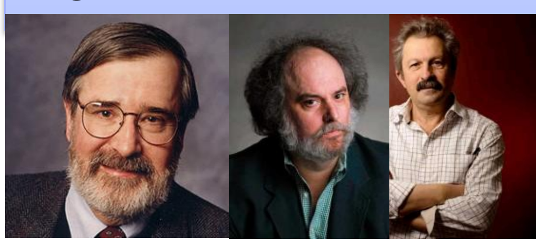
- … fully automatic
- … (reasonably) easy to use
- … provide (measurable) guarantees
- … come with guidelines and methodologies to apply effectively
- … apply to real software systems

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

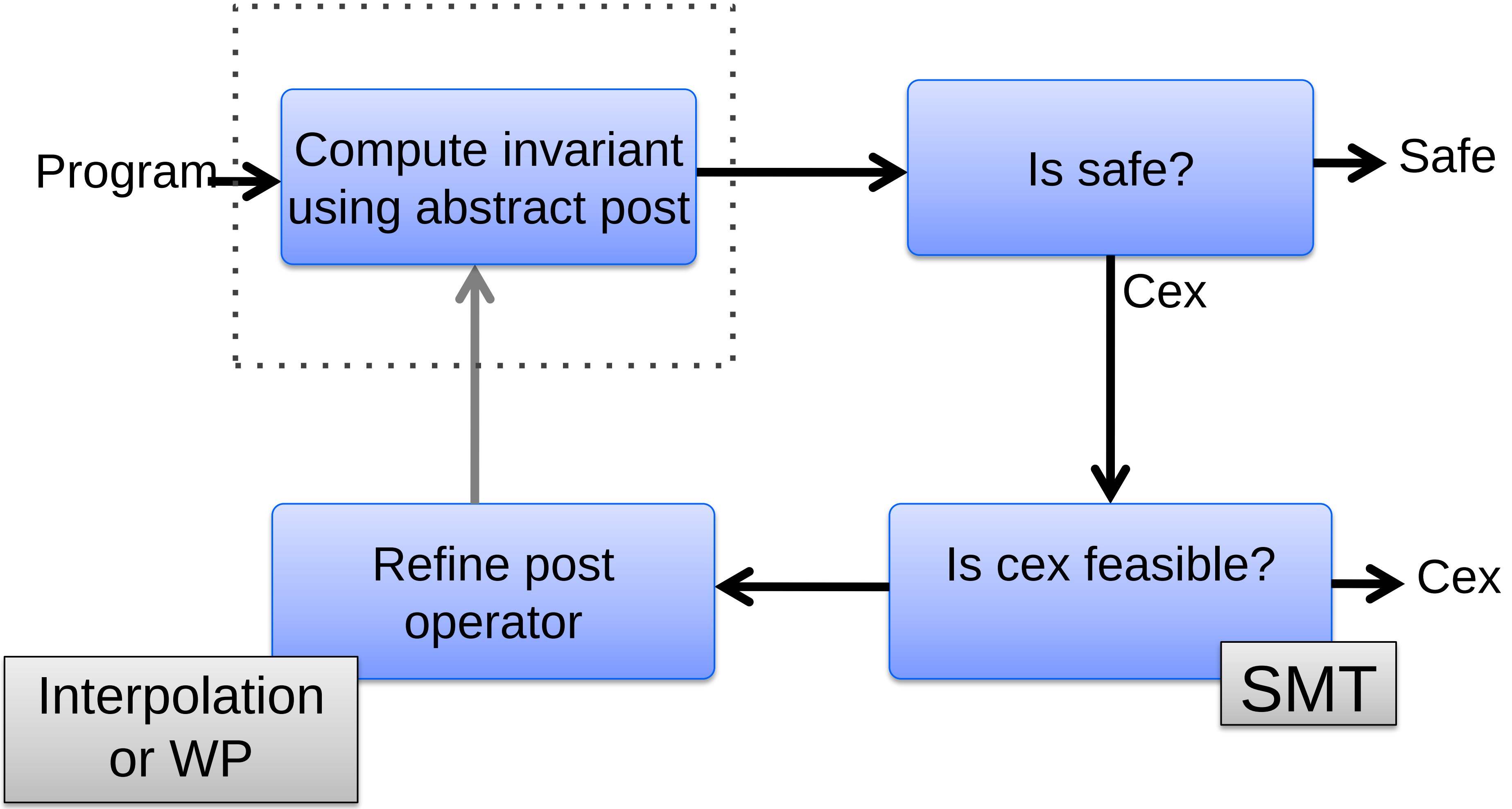
UFO: From Under- to Over- and Back!

Exploration Strategy

Refinement Strategy

Conclusion

Overapproximation-driven Approach (CEGAR)



e.g., BLAST, SLAM, CPAChecker, YaSM, SATAbs, etc.

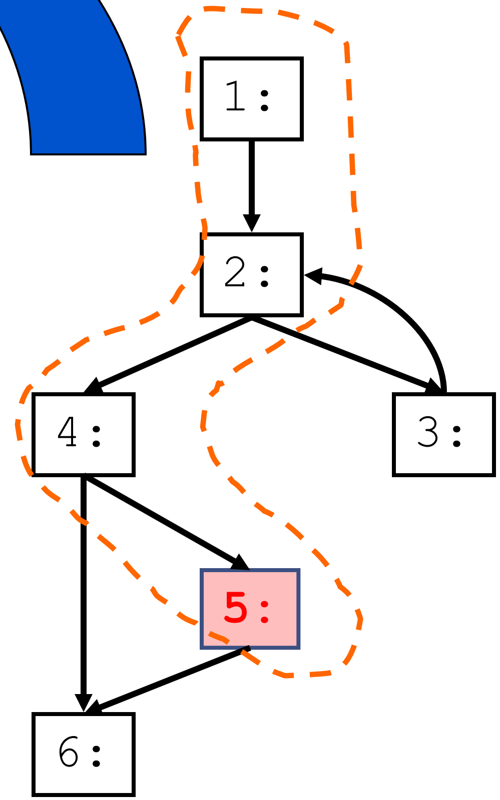
Is ERROR Reachable?

Program | **Abstraction** | **Over-Approximation**

Need This! (pointing to `(y <= 2)`)

```
1: int x = 2;
   int y = 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

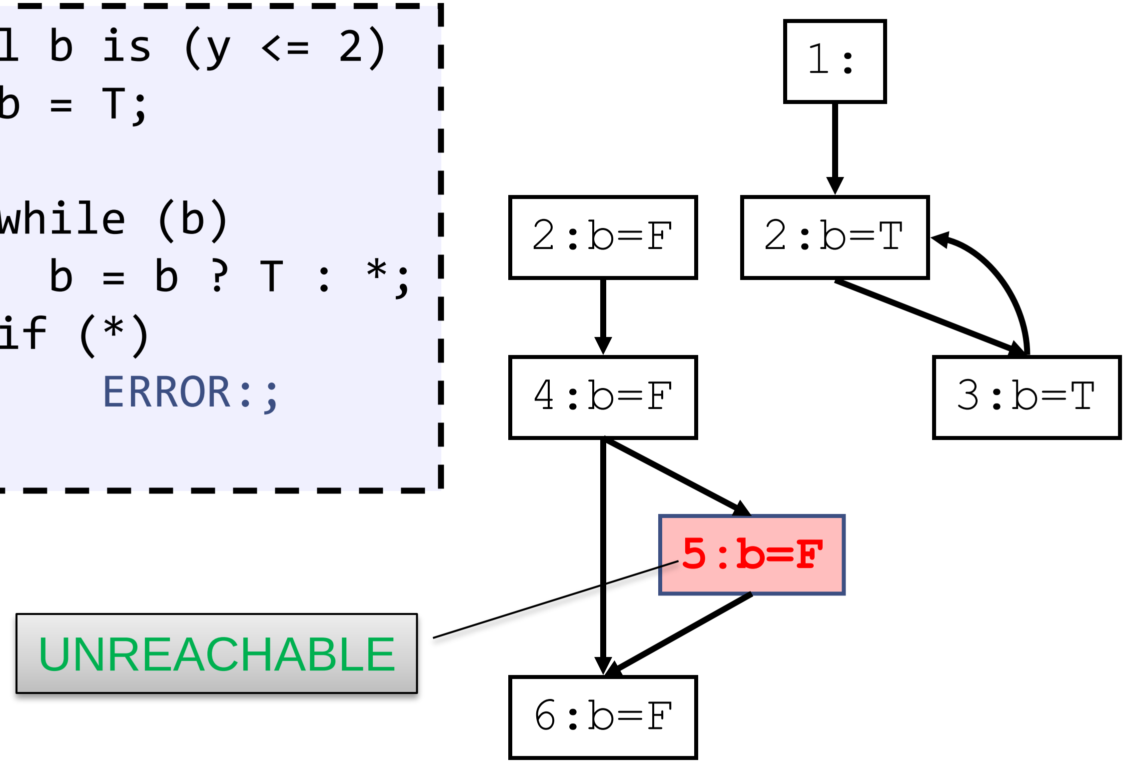
```
1: int x = 2;
   int y = 2;
2: while (y <= 2)
3:    y = y - 1;
4: if (x == 2)
5:       ERROR:;
6:
```

Abstraction

(with **y<=2**)

```
bool b is (y <= 2)
1: b = T;

2: while (b)
3:    b = b ? T : *;
4: if (*)
5:       ERROR:;
6:
```

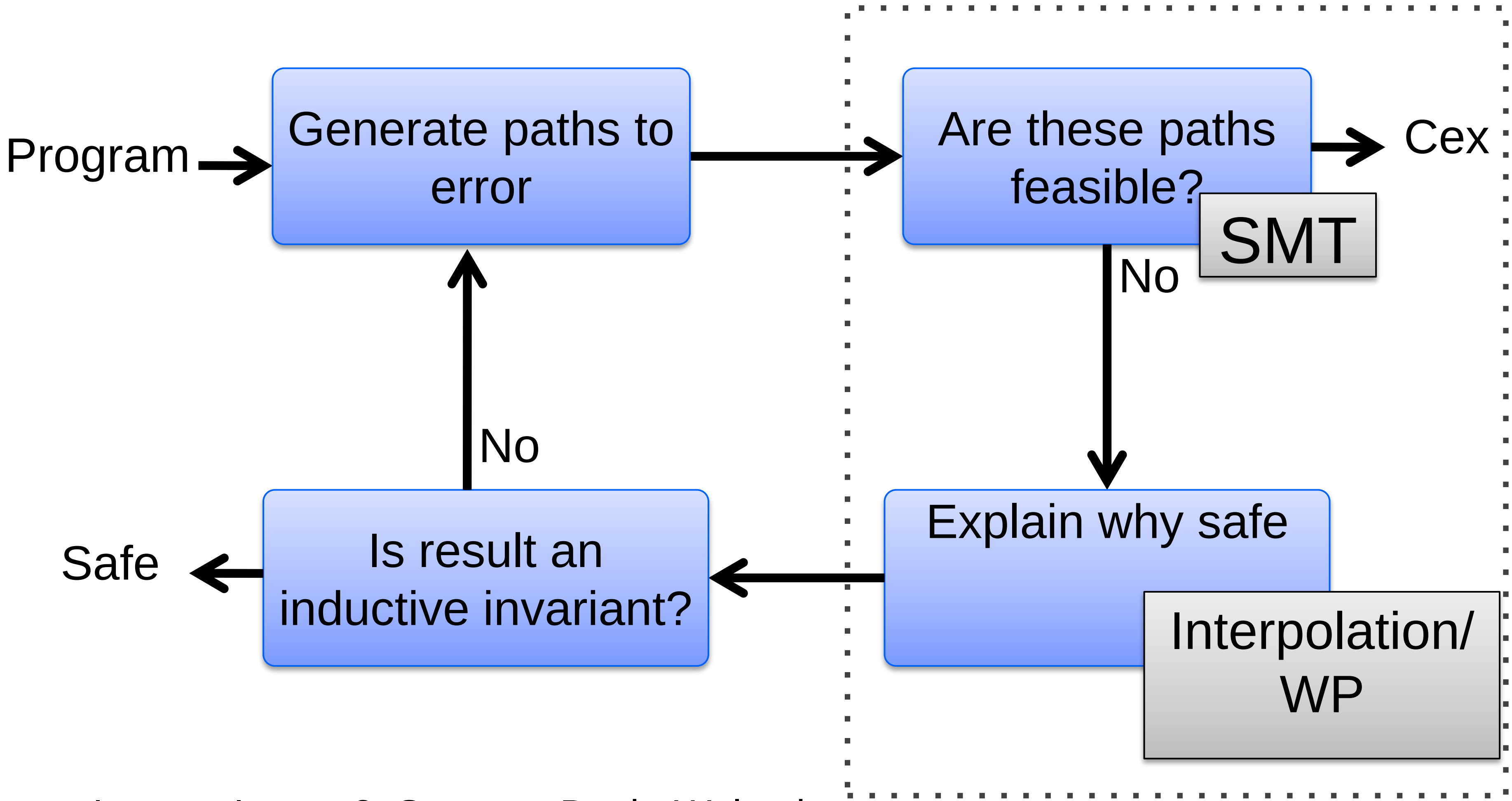
Over-Approximation



CEGAR steps

Abstract → Translate → Check → NO ERROR

Underapproximation-driven Approach (Impact)

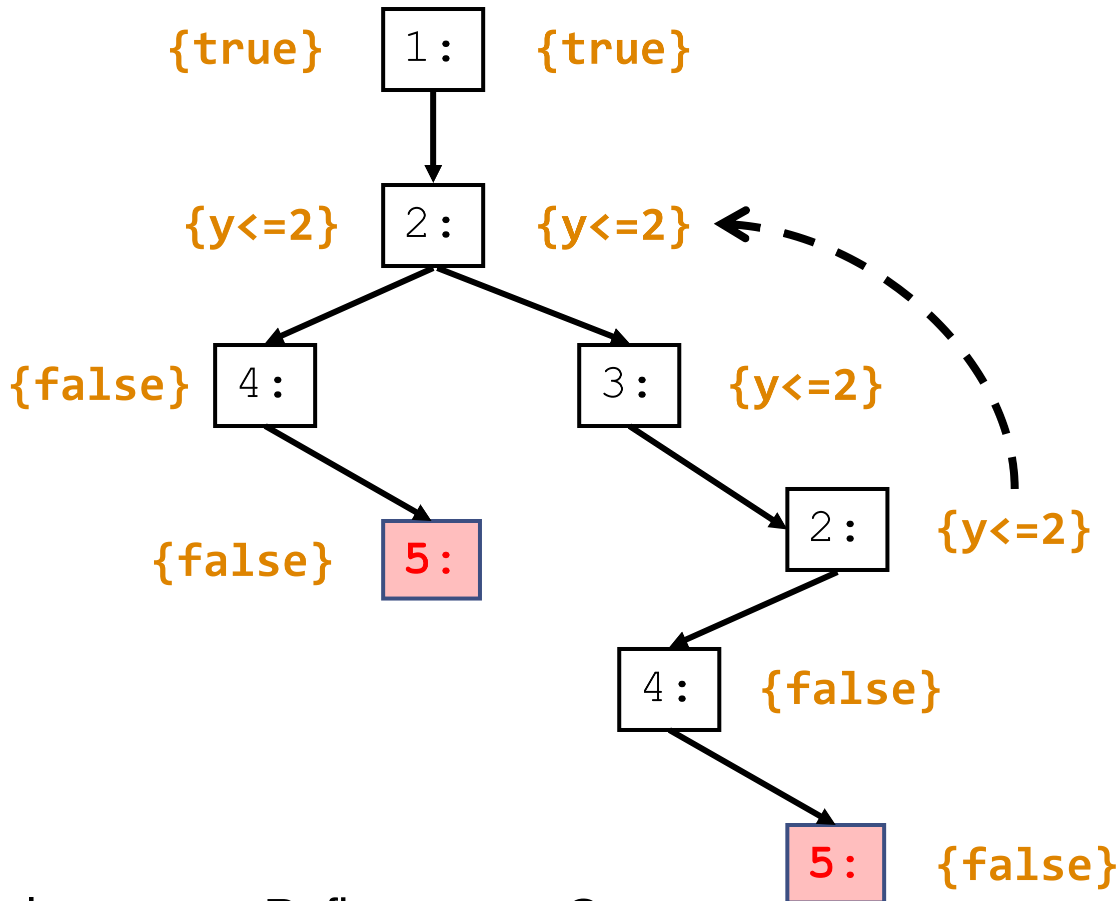


e.g., Impact, Impact2, Synergy, Dash, Wolverine

Under- Driven: Is ERROR Reachable?

Program

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



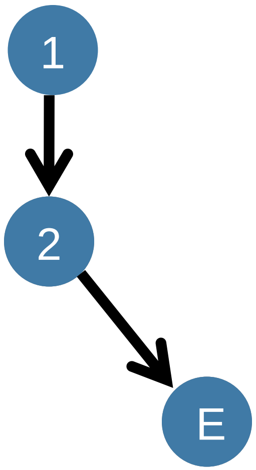
IMPACT steps

Explore → Refine → Explore → Refine → Cover

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

OD

UD




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

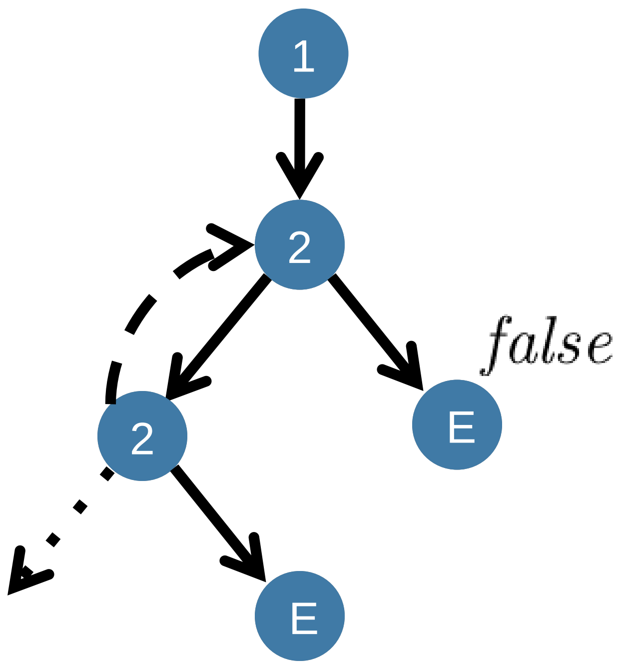
- Unlabeled
- Pred. abs. label
- Interpolant label

Explore
Refine
Explore

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

OD

UD




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

- Unlabeled
- Pred. abs. label
- Interpolant label

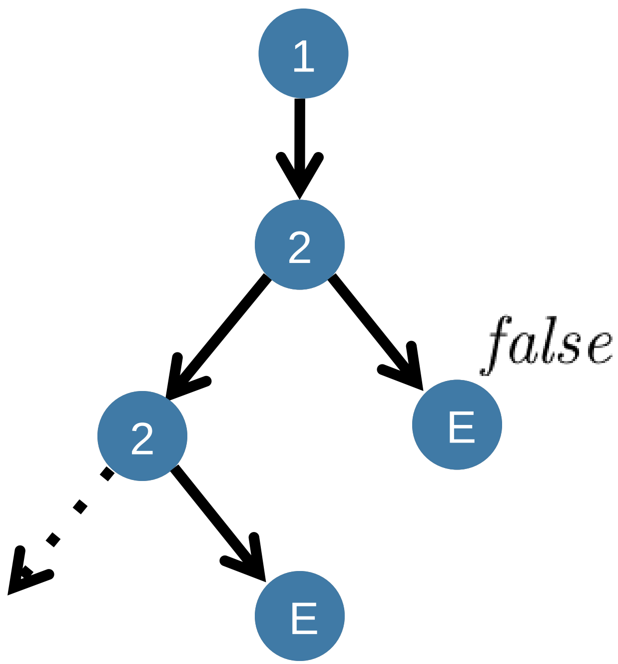
Explore

Refine

Explore

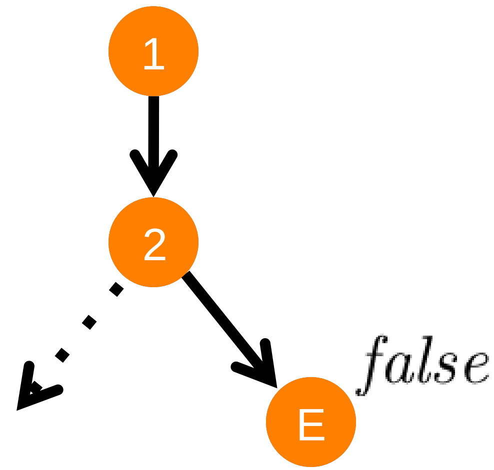
Over- Driven v.s. Under- Driven in a Nutshell

OD




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

UD

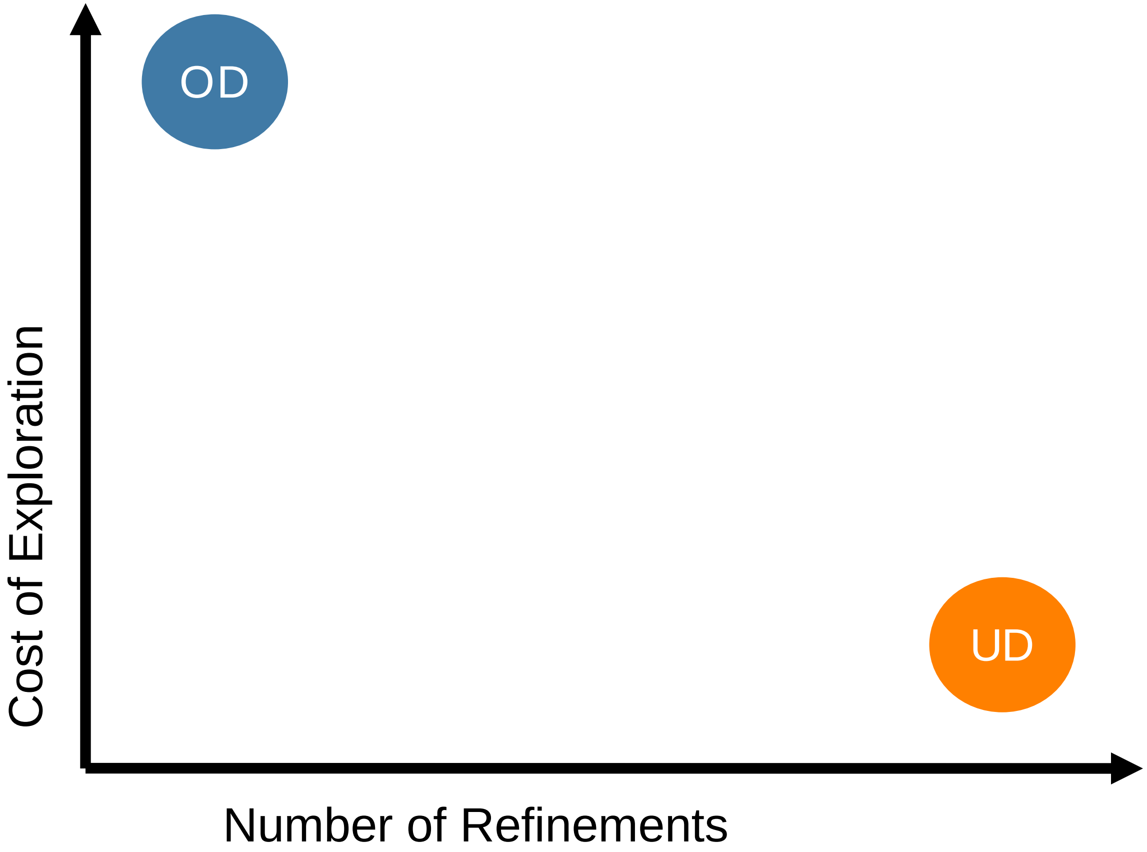


Explore
Refine
Explore

- Unlabeled
- Pred. abs. label
- Interpolant label

Explore
Refine
Explore

OD vs. UD Approaches



Our Algorithm: UFO

UD algorithm

Interpolation-based

OD algorithm

Predicate abstraction based

Combination of UD and OD

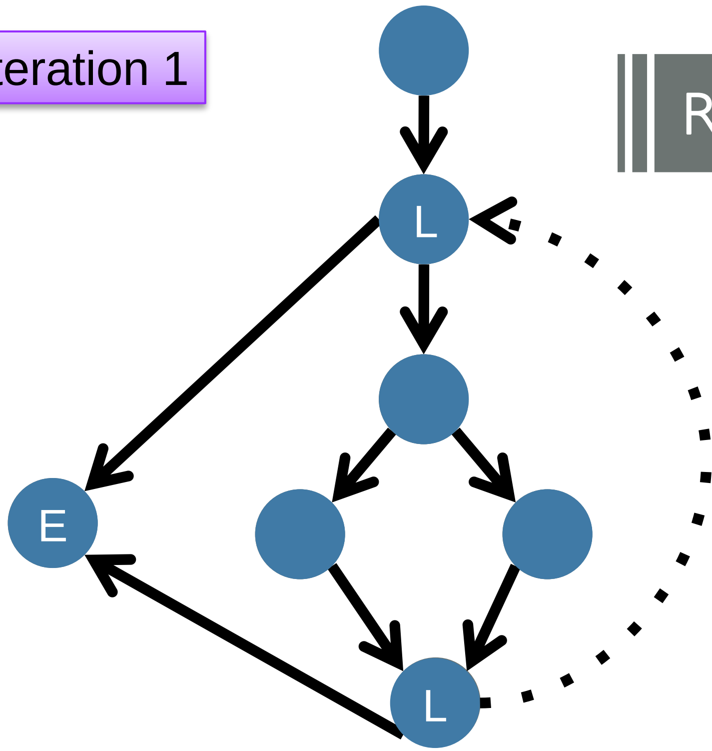
+

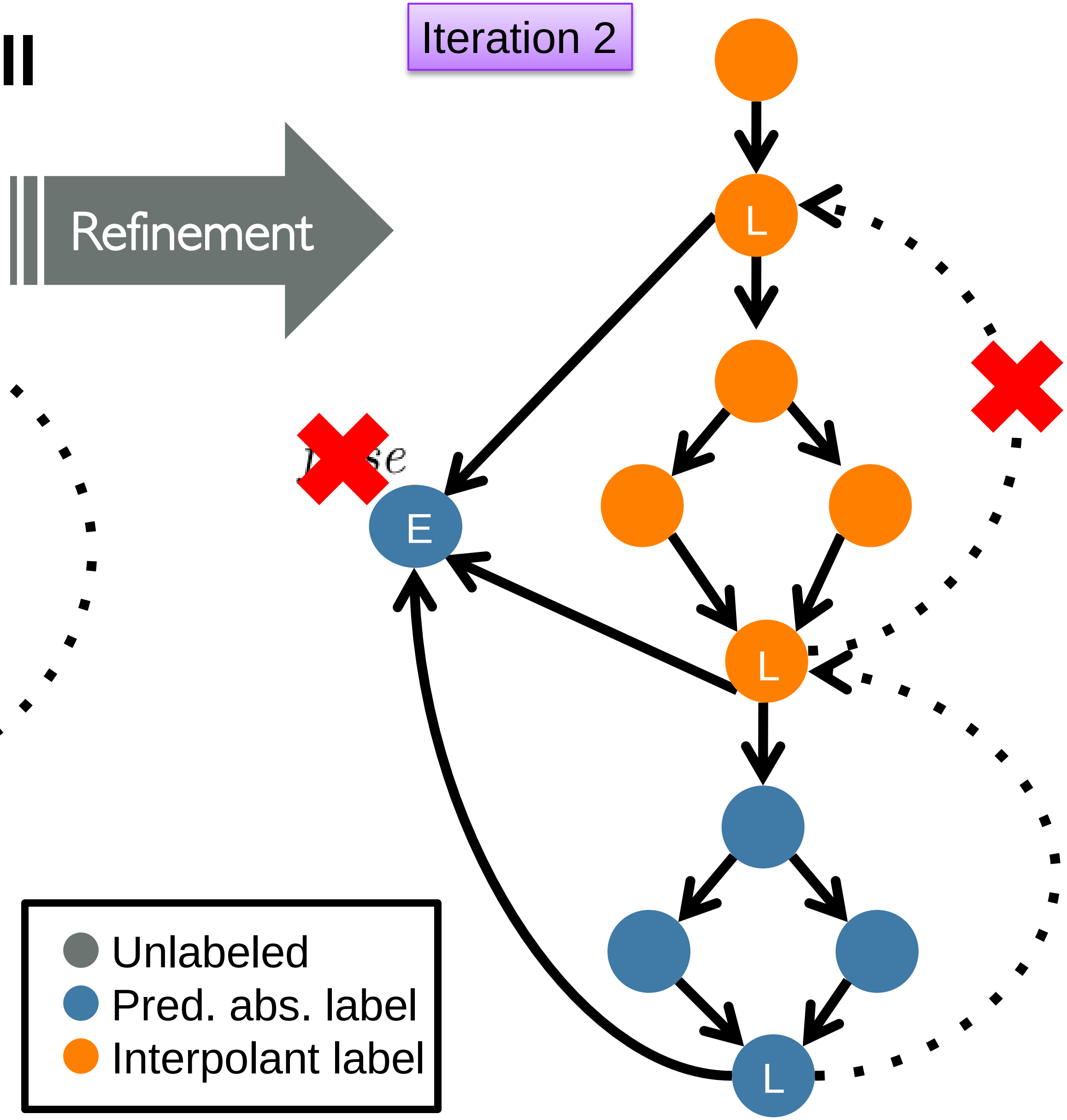
A novel interpolation-based refinement

Multiple paths checked and refined with a single SMT call

UFO in a Nutshell

Iteration 1





Imprecise post → UD
Explore from root → OD

- Unlabeled
- Pred. abs. label
- Interpolant label

The UFO Algorithm

```
func UfoMain (Program P) :
    create node v_en
    ψ(v_en) ← true, ν(v_en) ←
    marked(v_en) ← true
    labels ← ∅
    while true do
        ExpandArg()
        if ψ(v_err) is UNSAT then
            return SAFE
        labels ← Refine()
        if labels = ∅ then
            return UNSAFE
        clear AH and FN

func GetFutureNode (ℓ ∈ L) :
    if FN(ℓ) exists then
        return FN(ℓ)
    create node v
    ψ(v) ← true; ν(v) ← ℓ
    FN(l) ← v
    return v

func ExpandNode (v ∈ V) :
    if v has children then
        for all (v, w) ∈ E do
            FN(ν(w)) ← w
    else
        for all (ν(v), T, ℓ) ∈ Δ do
            w ← GetFutureNode(ℓ)
            E ← E ∪ {(v, w)}; τ(v, w) ← T

func ExpandArg () :
    v ← v_en
    while true do
        ExpandNode(v)
        if marked(v) then
            marked(v) ← false
            ψ(v) ← ⋁_{(u,v)∈E} Post(u, v)
            for all (v, w) ∈ E do marked(w) ← true
        else if labels(v) bound then
            ψ(v) ← labels(v)
            for all {(v, w) ∈ E | labels(w) unbound} do
                marked(w) ← true
        if v = v_err then break
        if ν(v) is head of a component then
            if ψ(v) ⇒ ⋁_{u∈AH(ν(v))} ψ(u) then
                erase AH(ν(v)) and FN(ν(v))
                l ← WtoExit(ν(v))
                v ← FN(l); erase FN(l)
                for all {(v, w) ∈ E | ∄u ≠ v · (u, w) ∈ E} do
                    erase FN(ν(w))
                continue
            add v to AH(ν(v))
        l ← WtoNext(ν(v))
        v ← FN(l); erase FN(l)
```

Explore

Refine

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:

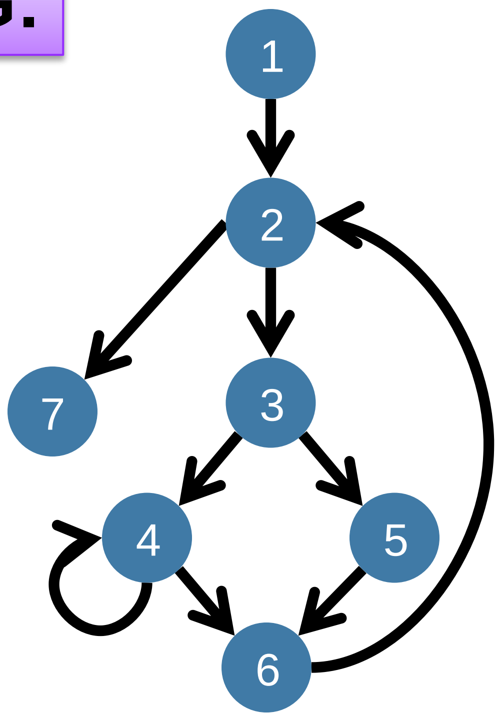
$$(u \prec v \wedge v \notin \omega(u)) \vee (u \preceq u \wedge v \in \omega(u))$$

Elements between two matching paren. are called *components*

First element of a component is called *head*

ω(u) is the set of heads of components containing u

DAG:

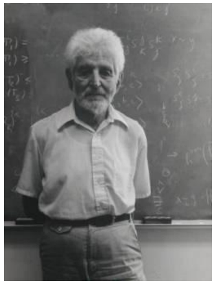


WTO:

(1 (2 3 (4) 5 6) 7)

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

$A \Rightarrow I$ $\quad\quad$ $I \Rightarrow \neg B$ $\quad\quad$ $\mathit{atoms}(I) \in \mathit{atoms}(A) \cap \mathit{atoms}(B)$

Theorem (McMillan 2003)

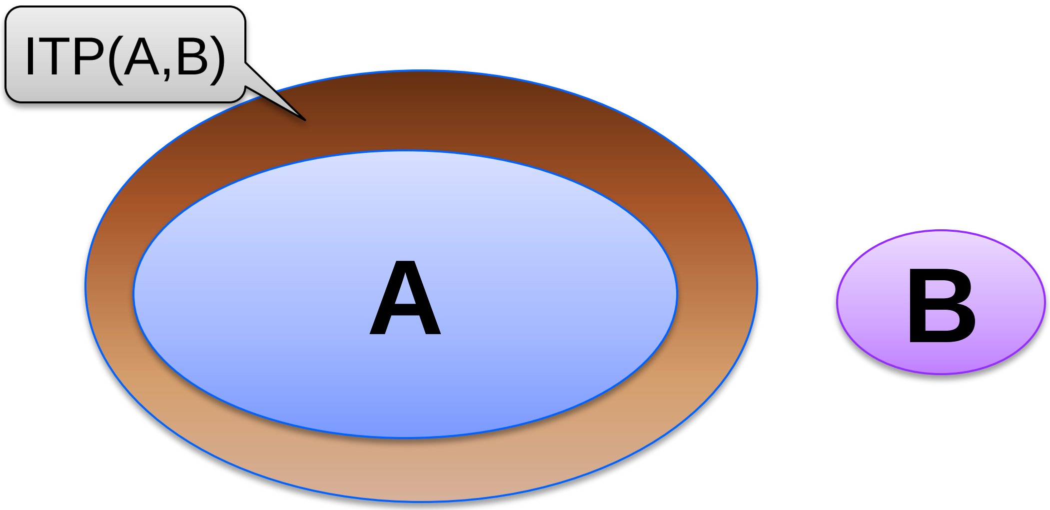
A Craig interpolant ITP(A, B) can be effectively constructed from a resolution proof of unsatisfiability of $A \wedge B$

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

Craig Interpolation in Model Checking

Over-Approximating Reachable States

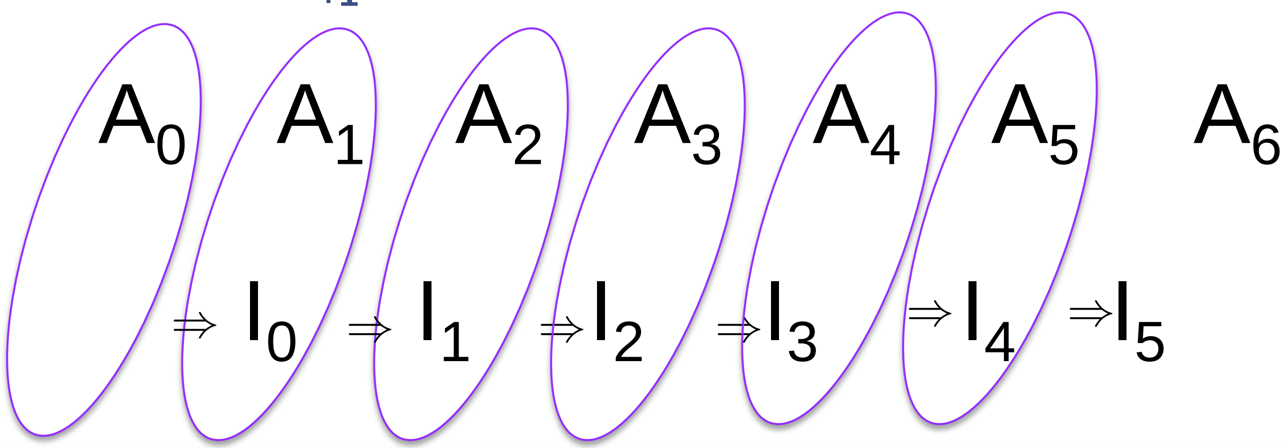
- Let $\mathbf{R^i}$ be the ith step of a transition system
- Let $\mathbf{A = Init} \land \mathbf{R^0} \land \ldots \land \mathbf{R^n}$ and $\mathbf{B = Bad}$
- **ITP (A, B)** (if exists) is an over-approx of states reachable in n-steps that does not contain any Bad states



Interpolation Sequence

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

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



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 π:E→Expr. A ***DAG Interpolant*** (if it exists) is a labeling I:V→Expr such that

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

$I_2 = ITP\ (\pi_1,\quad \pi_8)$

$I_2 = ITP\ (\pi_1,\quad \pi_2 \wedge \pi_3 \wedge \pi_6 \wedge \pi_7)$

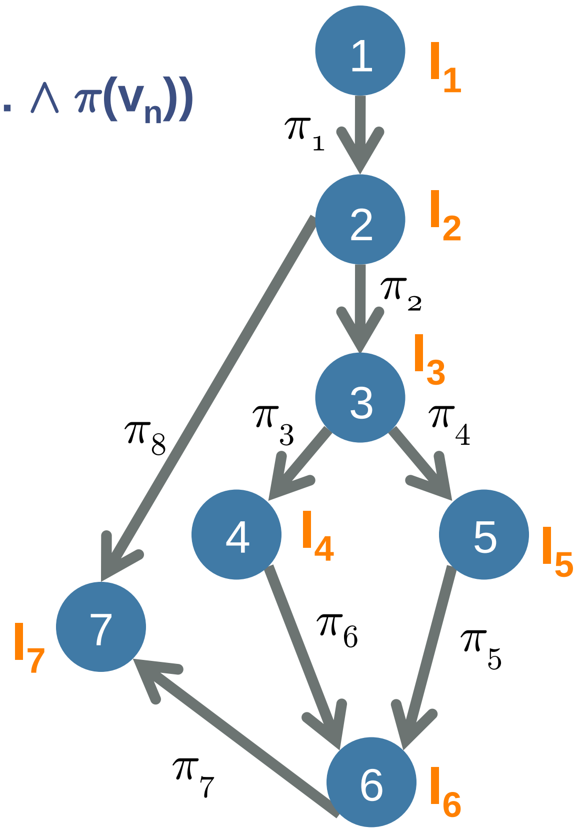
…

$(I_1 \wedge \pi_1) \Rightarrow I_2$

$(I_2 \wedge \pi_8) \Rightarrow I_7$

$(I_2 \wedge \pi_2) \Rightarrow I_3$

…



DAG Interpolation Algorithm

Reduce DAG Interpolation to Sequence Interpolation!

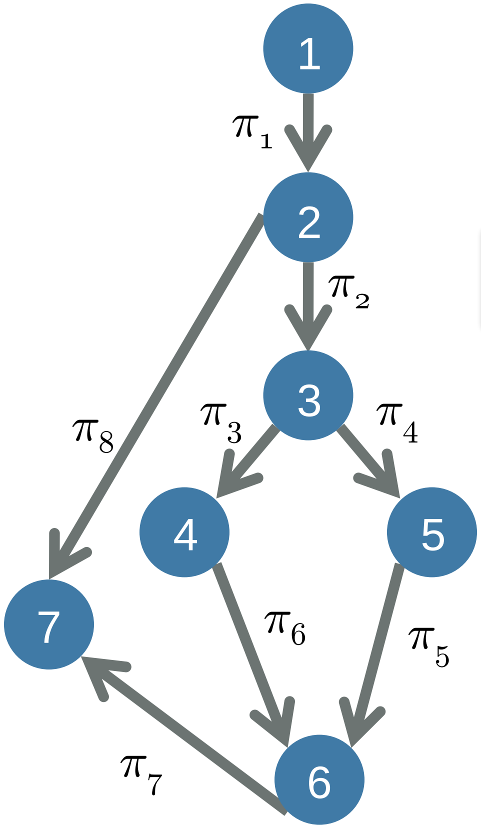
```
DagItp ((V, E), π)
{
   (A_0, …, A_n) = Encode(V, E, π)

   (I_1, …, I_{n-1}) = SeqItp(A_0, …, A_n)

   for i in [1, n-1] do J_i = Clean(I_i)

   return (J_1, …, J_{n-1})
}
```

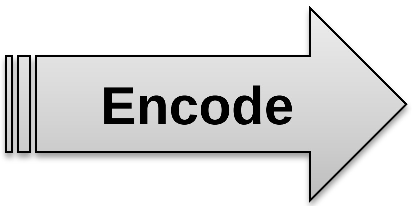
Encode input DAG by a set of constraints. One constraint per vertex.

Compute interpolant sequence. One interpolant per vertex.

Remove out-of-scope variables

DagItp: Encode





A_1: v_1; $v_1 \Rightarrow v_2 \wedge \pi_1$

A_2: $v_2 \Rightarrow (v_3 \wedge \pi_2) \vee (v_7 \wedge \pi_8)$

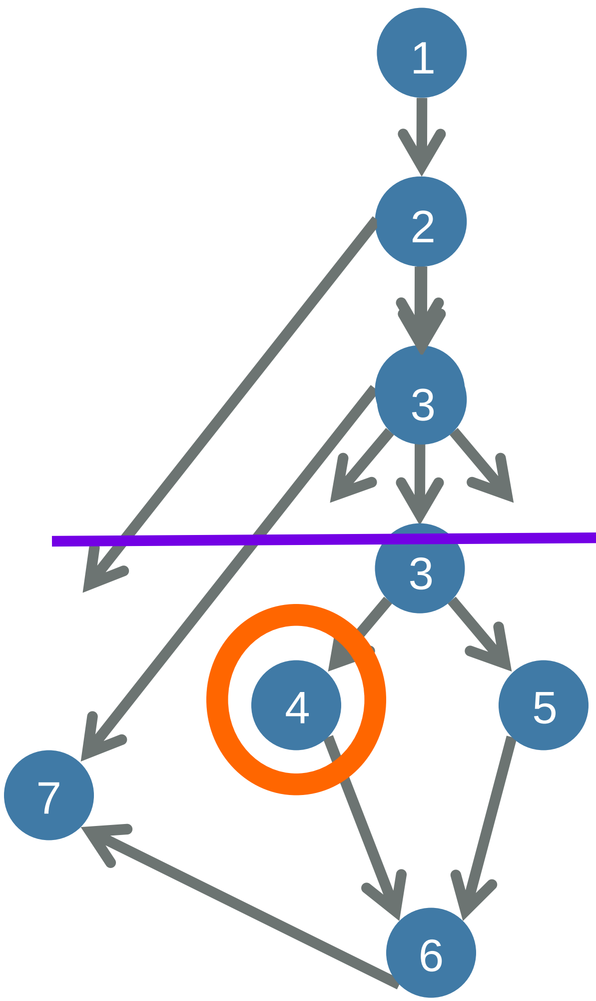
A_3: $v_3 \Rightarrow (v_4 \wedge \pi_3) \vee (v_5 \wedge \pi_4)$

A_4: $v_4 \Rightarrow v_6 \wedge \pi_6$

A_5: $v_5 \Rightarrow v_6 \wedge \pi_5$

A_6: $v_6 \Rightarrow v_7 \wedge \pi_7$

DagItp: Sequence Interpolate



A_1: v_1, $v_1 \Rightarrow v_2 \wedge \pi_1$

A_2: $v_2 \Rightarrow (v_3 \wedge \pi_2) \vee (v_7 \wedge \pi_8)$

A_3: $v_3 \Rightarrow (v_4 \wedge \pi_3) \vee (v_5 \wedge \pi_4)$

I_4

A_4: $v_4 \Rightarrow v_6 \wedge \pi_6$

A_5: $v_5 \Rightarrow v_6 \wedge \pi_5$

A_6: $v_6 \Rightarrow v_7 \wedge \pi_7$

DagItp: Clean

`Clean(I_i) =`

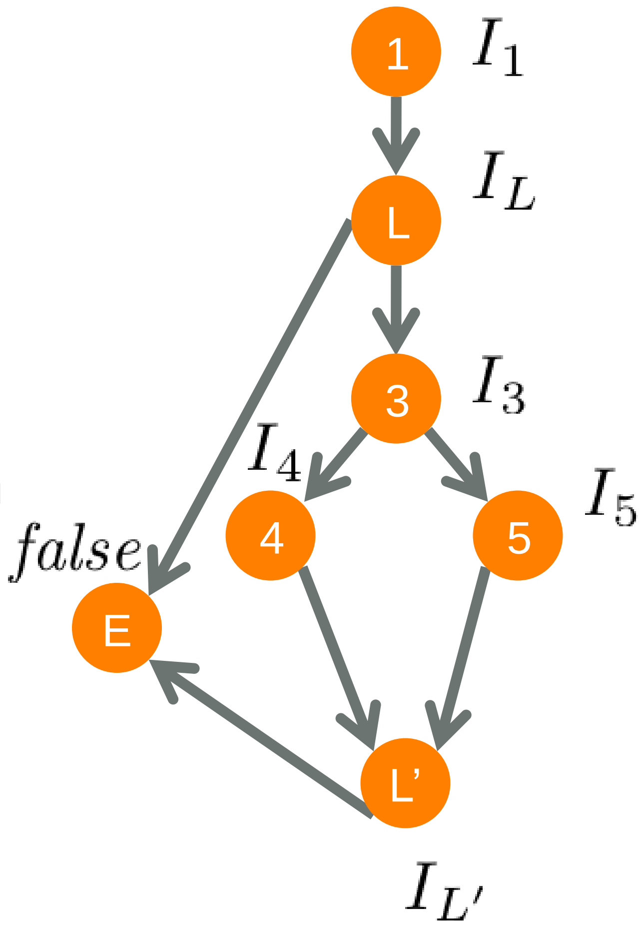
$$\forall\{x \mid x \in \mathrm{var}(I_i) \wedge \neg \mathrm{inScope}(x, v_i)\} \cdot \forall\{v_j \mid v_j \in V\} \cdot I[v_i \leftarrow \top]$$

UFO Refinement

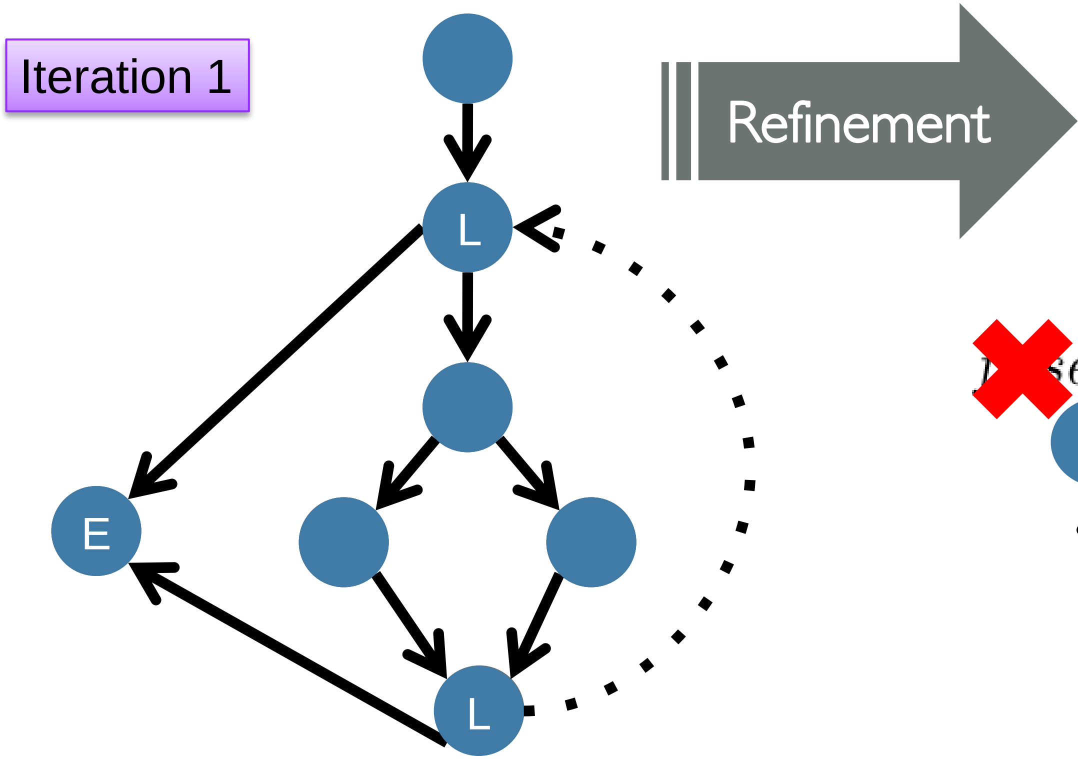
1. Construct DAG of current unfolding
2. Use DagItp to find new labels

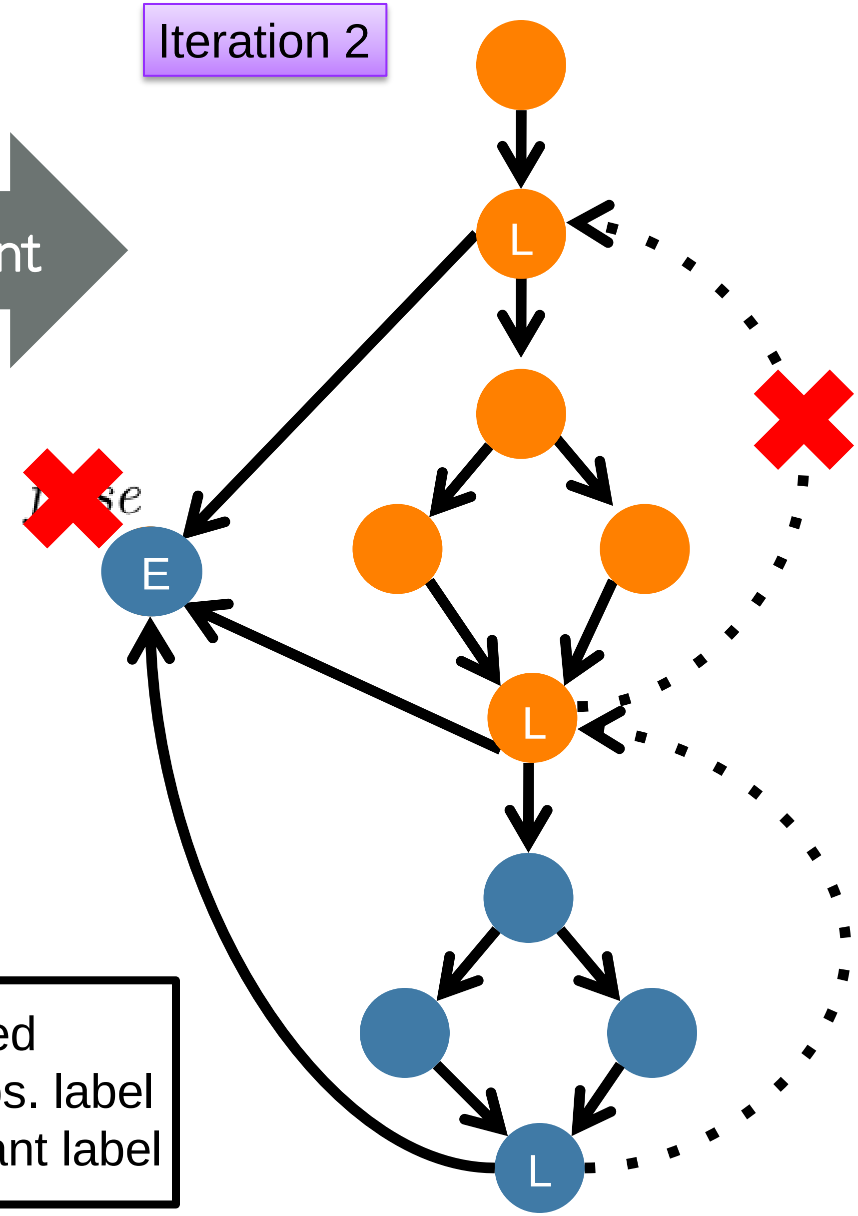
Refinement is done with a ***single*** SMT call

Cleaning the labels with quantifier elimination is a major bottleneck



UFO in a Nutshell

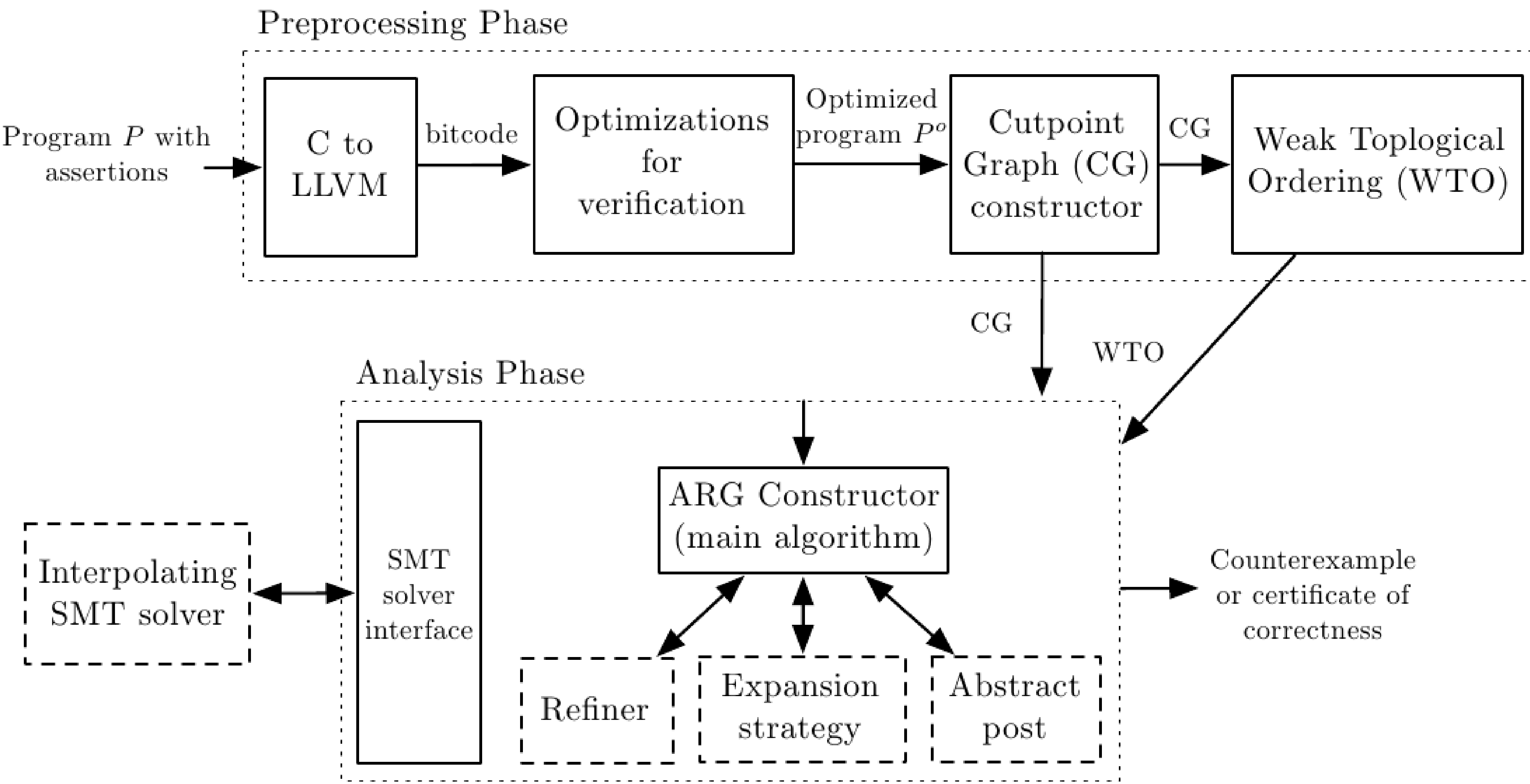




Imprecise post → UD
Explore from root → OD

- Unlabeled
- Pred. abs. label
- Interpolant label

UFO Framework: Architecture



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-simplifed, 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

	#SOLVED	#SAFE	#UNSAFE	TOT. TIME (s)
ufoNo	78	22	56	8,289
ufoCP	**79**	**22**	**57**	**7,838**
ufoBP	69	17	52	11,260
CP	49	10	39	15,363
BP	71	19	52	10,018
Wolverine	38	18	20	19,753

Results: A Closer Look (SAFE)

	ufoNo		ufoCP		ufoBP		BP	
	TIME	#REF	TIME	#REF	TIME	#REF	TIME	#REF
token1	98	18	24	10	**0.69**	4	0.69	4
token2	--	--	--	--	**2.15**	4	2.63	4
token3	--	--	--	--	**76**	4	--	--
token4	--	--	--	--	--	--	**153**	4
token5	--	--	--	--	--	--	**149**	4
srvr1a	5.2	10	5.16	8	0.79	4	**0.43**	3
srvr1b	1.37	7	2.9	7	**0.89**	5	--	--
srvr2	**171**	17	184	17	--	--	--	--
srvr3	133	17	147	17	--	--	**33.71**	5
srvr4	--	--	--	--	--	--	**8**	4
srvr8	101	14	115	14	--	--	--	--

Results: A Closer Look (UNSAFE)

	ufoNo		ufoCP		ufoBP		BP	
	TIME	#REF	TIME	#REF	TIME	#REF	TIME	#REF
kundu1	--	--	**24**	4	122	4	33	3
kundu2	**1.24**	2	2.74	2	8.15	2	8.6	2
toy1	96	10	79	9	**13.54**	3	--	--
toy2	**12**	5	60	8	--	--	--	--
token12	27	4	**14**	4	--	--	--	--
token13	37	4	**34**	4	--	--	--	--
token14	**10**	3	33	4	--	--	--	--
token15	52	4	34	4	--	--	--	--

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

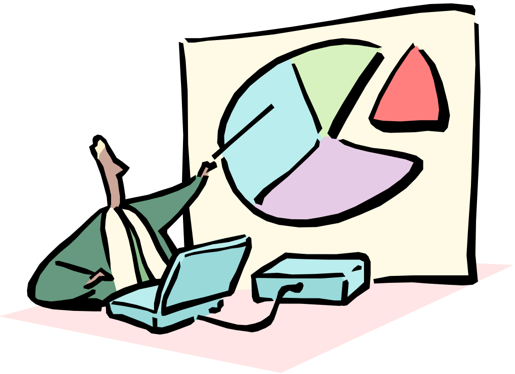
Whale [Our work 12]

- 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

U.S. mail:

Software Engineering Institute

Customer Relations

4500 Fifth Avenue

Pittsburgh, PA 15213-2612

USA

Web:

www.sei.cmu.edu

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

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