Loop Summarization using Abstract Transformers

Daniel Kröning[1], Natasha Sharygina[2,5], Stefano Tonetta[3], **Aliaksei Tsitovich**[1] and Christoph M. Wintersteiger[4]

[1]Oxford University, Computing Laboratory, UK

[1] University of Lugano, Switzerland

[3]Fondazione Bruno Kessler, Trento, Italy

[4]Computer Systems Institute, ETH Zurich, Switzerland

[5]School of Computer Science, Carnegie Mellon University, USA

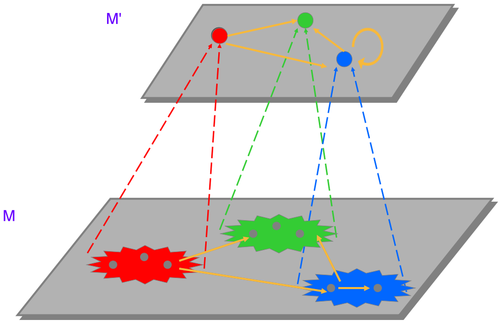
October 21, 2008

What am I going to present in the next 20 minutes?

- Idea of loop summarization
- LoopFrog - tool, which implements it all

Abstraction

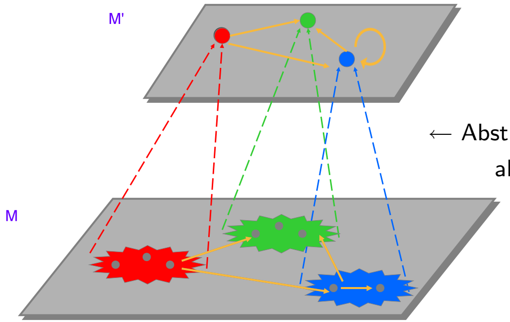
Software static analysis needs abstraction



All successful techniques use abstraction

Abstraction

Software static analysis needs abstraction



← Abstraction function α (e.g., using abstract domains or predicates)

All successful techniques use abstraction

Abstract Interpretation

- Define abstract domains (e.g., intervals, polyhedra)
- Iteratively evaluate the program until fixpoint is reached

Abstract Interpretation

- Define abstract domains (e.g., intervals, polyhedra)
- Iteratively evaluate the program until fixpoint is reached

Problem 1

Iterative fixpoint computation is EXPENSIVE

Also, in case of failure, no counter-example is given as feedback.

Existing Solution

- To make iterative computation converge - apply widening (overapproximation of the set of abstract values)

Existing Solution

- To make iterative computation converge - apply widening (overapproximation of the set of abstract values)

Problem 2

Widening causes Imprecision

Introduces a lot of false positives

Tools: Polyspace, ASTRÉE

Alternative approach: CEGAR-based techniques

- Abstract the program according to set of the predicates.
- Check the abstract model and get the counter-example (CE)
- If CE is spurious - refine set of predicates to remove it, update the abstract model and repeat iteratively until a real CE or no CEs at all.

Alternative approach: CEGAR-based techniques

- Abstract the program according to set of the predicates.
- Check the abstract model and get the counter-example (CE)
- If CE is spurious - refine set of predicates to remove it, update the abstract model and repeat iteratively until a real CE or no CEs at all.

Problem 3

Abstraction requires either quantification or overapproximation

First variant blows up. Second introduces spurious transitions.

Tools: SATABS, SLAM, BLAST, MAGIC etc.

Loops

The common problem of iterative fixpoint computations

They All Are Afraid Of Loops!

How would they handle this?

Example

```
p=a;
while(*p!=0){
  if(*p=='/')
    *p=0;
  p++;
}
```

How would they handle this?

Example

```
p=a;
while(*p!=0){
  if(*p=='/')
    *p=0;
  p++;
}
```

- CEGAR-based → try to get predicates until it fails (might not terminate)

How would they handle this?

Example

```
p=a;
while(*p!=0){
  if(*p=='/')
    *p=0;
  p++;
}
```

- CEGAR-based → try to get predicates until it fails (might not terminate)
- Abs. Int. → (precise domain + aggressive widening) or imprecise domain

Our Solution

- *Avoid* iterative computation of an abstract fixpoint. Instead build summaries. Make the summaries *precise*.
  - Encode loop-free fragments into concrete summaries.
  - Replace each loop by its abstract summary.
- Perform an assertion check on the obtained abstract model. Since there are no loops anymore, iterative computation is avoided.

Our Solution

- *Avoid* iterative computation of an abstract fixpoint. Instead build summaries. Make the summaries *precise*.
  - Encode loop-free fragments into concrete summaries.
  - Replace each loop by its abstract summary.

  (I will explain how to construct an abstract summary on example)
- Perform an assertion check on the obtained abstract model. Since there are no loops anymore, iterative computation is avoided.

Prepare the loop for summarization

Example

```
p=a;
while(*p!=0){
  if(*p=='/')
    *p=0;
  p++;
}
```

Transformer of the loop guard

$$(((*p == 0) \wedge z_a \wedge (p_a \geq l_a)) \\ \vee((*p! = 0) \wedge (p_a \neq l_a) \wedge \ldots))$$

Transformer of the loop body

$$((*p =' /' \wedge a' = a[*p = 0]) \\ \vee(*p \neq' /' \wedge a' = a)) \\ \wedge(p' = p + 1)$$

- p_a - offset of the pointer p from the base address of the array a
- z_a - *True* if a contains the zero character
- s_a - *True* if a contains the slash character
- l_a is the index of the first zero character (if present).[1]

---

[1] l_a, z_a and b_a (buffer size) are instrumented according to Dor et. al.

Invariant candidates

Heuristically provide invariant candidates ψ to use as a summary:

- $(0 \leq p_a \leq l_a) \wedge z_a \wedge \neg s_a$ - pointer offset is bounded by string length and doesn't contain slash character
- z_a - string remains zero-terminated

Single loop summarization

Example

```
p=a;
while(*p!=0){
  if(*p=='/')
    *p=0;
  p++;
}
```

To every candidate assertion ψ we apply:

1. transformer of the loop guard
2. transformer of the loop body

If obtained $\psi' \implies \psi$ then ψ is invariant of the loop (implication is checked using a decision procedure, e.g., SAT).

Loop summary

$$a'[] = nondet() \land p'_a = nondet() \land (z'_a = z_a)$$
$$\land((0 \leq p'_a \leq l_a) \land z'_a \land \neg s_a)$$

Summary, i.e. symbolic transformer, is constructed, not iteratively computed

Summarization for arbitrary programs

```
SUMMARIZE(π)
input     : program π = ⟨U, G⟩
output    : over-approximation π′ of π
begin
    ⟨T, >⟩ :=sub-graph dependency tree of π;
    π_r := π;
    for each G′ such that G > G′ do
        ⟨U, G′′⟩:=SUMMARIZE(⟨U, G′⟩);
        π_r := π_r where G′ is replaced with G′′;
        update ⟨T, >⟩;
    if π_r is a single loop then
        ⟨Â, t⟩ := choose abstract interpretation for π_r;
        ψ := test invariant candidates for t on π_r;
        π′ := SINGLELOOPSUMMARY(π_r, Â, t_ψ);
    else
        /* π_r is loop-free */
        π′ := Sum_⟨A,τ⟩(π_r);
    return π′
end
```

1. Call summarization recursively for all nested loops
2. Construct abstract summary for each single loop
3. Encode concrete summary of the loop-free fragment

Summarization for arbitrary programs

$\textsc{Summarize}(\pi)$
input : program $\pi = \langle U, G \rangle$
output : over-approximation π' of π
begin
$\langle T, > \rangle$:=sub-graph dependency tree of π;
$\pi_r := \pi$;
for each G' **such that** $G > G'$ **do**
$\langle U, G'' \rangle := \textsc{Summarize}(\langle U, G' \rangle)$;
$\pi_r := \pi_r$ where G' is replaced with G'';
update $\langle T, > \rangle$;
if π_r **is a single loop then**
$\langle \hat{A}, t \rangle$:= choose abstract interpretation for π_r;
ψ := test invariant candidates for t on π_r;
$\pi' := \textsc{SingleLoopSummary}(\pi_r, \hat{A}, t_\psi)$;
else
/* π_r is loop-free */
$\pi' := Sum_{\langle A, \tau \rangle}(\pi_r)$;
return π'
end

- Linear in number of loops

Summarization for arbitrary programs

SUMMARIZE(π)
input : program $\pi = \langle U, G \rangle$
output : over-approximation π' of π
begin
 $\langle T, > \rangle$:=sub-graph dependency tree of π;
 $\pi_r := \pi$;
 for each G' **such that** $G > G'$ **do**
  $\langle U, G'' \rangle$:=SUMMARIZE($\langle U, G' \rangle$);
  $\pi_r := \pi_r$ where G' is replaced with G'';
  update $\langle T, > \rangle$;
 if π_r **is a single loop then**
  $\langle \hat{A}, t \rangle$:= choose abstract interpretation for π_r;
  ψ := test invariant candidates for t on π_r;
  π' := SINGLELOOPSUMMARY(π_r, $\hat{A}$, t_ψ);
 else
  /* π_r is loop-free */
  $\pi' := Sum_{\langle A, \tau \rangle}(\pi_r)$;
 return π'
end

- Linear in number of loops
- Summarization of each loop takes **finite** number of calls to decision procedure.

Summarization for arbitrary programs

SUMMARIZE(π)
input : program $\pi = \langle U, G \rangle$
output : over-approximation π' of π
begin
$\langle T, > \rangle$:=sub-graph dependency tree of π;
$\pi_r := \pi$;
for each G' **such that** $G > G'$ **do**
$\langle U, G'' \rangle$:=SUMMARIZE($\langle U, G' \rangle$);
$\pi_r := \pi_r$ where G' is replaced with G'';
update $\langle T, > \rangle$;
if π_r **is a single loop then**
$\langle \hat{A}, t \rangle :=$ choose abstract interpretation for π_r;
$\psi :=$ test invariant candidates for t on π_r;
$\pi' :=$ SINGLELOOPSUMMARY($\pi_r, \hat{A}, t_\psi$);
else
/* π_r is loop-free */
$\pi' := Sum_{\langle A, \tau \rangle}(\pi_r)$;
return π'
end

- Linear in number of loops
- Summarization of each loop takes **finite** number of calls to decision procedure.
- Precision depends on selection of abstract domains

Summarization for arbitrary programs

```
SUMMARIZE(π)
input    : program π = ⟨U, G⟩
output   : over-approximation π′ of π
begin
    ⟨T, >⟩ :=sub-graph dependency tree of π;
    π_r := π;
    for each G′ such that G > G′ do
        ⟨U, G′′⟩:=SUMMARIZE(⟨U, G′⟩);
        π_r := π_r where G′ is replaced with G′′;
        update ⟨T, >⟩;
    if π_r is a single loop then
        ⟨Â, t⟩ := choose abstract interpretation for π_r;
        ψ := test invariant candidates for t on π_r;
        π′ := SINGLELOOPSUMMARY(π_r, Â, t_ψ);
    else
        /* π_r is loop-free */
        π′ := Sum_⟨A,τ⟩(π_r);
    return π′
end
```

- Linear in number of loops
- Summarization of each loop takes **finite** number of calls to decision procedure.
- Precision depends on selection of abstract domains
- Abstract domains are localized to loops

Assertion check

Model checker is used to check the assertions on the obtained loop-less model.

As a feedback user gets:

- Path (partial) to a violated assertion with variables assignment, i.e. *leaping counter-example*
- Results of summarization along the path
  - Loop summary and original loop body
  - Applied abstract domains
  - Discovered invariants
  - Rejected invariants

Implementation

LoopFrog- static analysis tool for C programs

- Models from C programs are created using Goto-CC front-end[2];
- Uses symbolic engine of CBMC for invariant candidates check and final assertion check.
- Currently doesn't support recursive calls.

[2] http://www.cprover.org/goto-cc

Abstract domains

Invariant	Details
$z_s = true$	Tests if zero-ermination is preserved
$z_s \wedge l_s < b_s$	Tests if string content stays within allocated buffer
$z_s \wedge 0 \leq i < l_s$	Tests if iterator value is bounded by string length
$0 \leq i < b_s$	Tests if iterator value is bounded by allocated buffer size
$valid_p = true$	Tests for pointer offset validity preservation

Table: Some of the domains in the LoopFrog's library.

Benchmark suite

	$R(d)$	$R(f)$	$R(\neg f\|d)$
Benchmark suite from Zitser et.al.			
LoopFrog	1.00	0.38	0.62
Interval Domain	1.00	0.98	0.02
Polyspace	0.87	0.50	0.37
Splint	0.57	0.43	0.30
Boon	0.05	0.05	0
Archer	0.01	0	0
Uno	0	0	0

Table: $R(d)$, $R(f)$ and $R(\neg f|d)$ for various static analysis tools.

- Detection rate $R(d)$ — number of correctly detected bugs
- False positive rate $R(f)$ — number of incorrectly detected bugs in fixed versions of test cases
- Discrimination rate $R(\neg f|d)$ — ratio of test cases on which an error is correctly reported, while it is, also correctly, not reported in the corresponding fixed test case.

Large-scale evaluation

Suite	Program	Instructions	# Loops	Time			Peak Memory	Assertions		
				Summarization	Checking Assertions	Total		Total	Passed	Violated
freecell-solver	aisleriot-board-2.8.12	347	26	10s	295s	305s	111MB	358	165	193
freecell-solver	gnome-board-2.8.12	208	8	0s	3s	4s	13MB	49	16	33
freecell-solver	microsoft-board-2.8.12	168	4	2s	9s	11s	32MB	45	19	26
freecell-solver	pi-ms-board-2.8.12	185	4	2s	10s	13s	33MB	53	27	26
gnupg	make-dns-cert-1.4.4	232	5	0s	0s	1s	9MB	12	5	7
gnupg	mk-tdata-1.4.4	117	1	0s	0s	0s	3MB	8	7	1
inn	encode-2.4.3	155	3	0s	2s	2s	6MB	88	66	22
inn	ninpaths-2.4.3	476	25	5s	40s	45s	49MB	96	47	49
ncompress	compress-4.2.4	806	12	45s	4060s	4106s	345MB	306	212	94
texinfo	ginstall-info-4.7	1265	46	21s	326s	347s	127MB	304	226	78
texinfo	makedoc-4.7	701	18	9s	6s	16s	28MB	55	33	22
texinfo	texindex-4.7	1341	44	415s	9336s	9757s	1021MB	604	496	108
wu-ftpd	ckconfig-2.5.0	135	0	0s	0s	0s	3MB	3	3	0
wu-ftpd	ckconfig-2.6.2	247	10	13s	43s	57s	27MB	53	10	43
wu-ftpd	ftpcount-2.5.0	379	13	10s	32s	42s	37MB	115	41	74
wu-ftpd	ftpcount-2.6.2	392	14	8s	24s	32s	39MB	118	42	76
wu-ftpd	ftprestart-2.6.2	372	23	48s	232s	280s	55MB	142	31	111
wu-ftpd	ftpshut-2.5.0	261	5	1s	9s	10s	13MB	83	29	54
wu-ftpd	ftpshut-2.6.2	503	26	27s	79s	106s	503MB	232	210	22
wu-ftpd	ftpwho-2.5.0	379	13	7s	23s	30s	37MB	115	41	74
wu-ftpd	ftpwho-2.6.2	392	14	8s	27s	35s	39MB	118	42	76
wu-ftpd	privatepw-2.6.2	353	9	4s	17s	22s	32MB	80	51	29

To conclude

- We proposed an algorithm for static analysis of programs which is:
    - sound (loop-less model is an overapproximation of the program)
    - scalable (avoids iterative abstract fixpoint computation)
    - precise (with configurable precision)
    - giving you feedback (leaping counter-example)

- We implemented it in a tool LoopFrog (http://www.verify.inf.unisi.ch/loopfrog)

- We applied LoopFrog to a wide range of benchmarks and the result shows that it outperforms the competitors

Thanks for listening!

Questions ?