Big-Step Bounded Model Checking for Software

Nishant Sinha,
IBM Research Labs
Bangalore, India

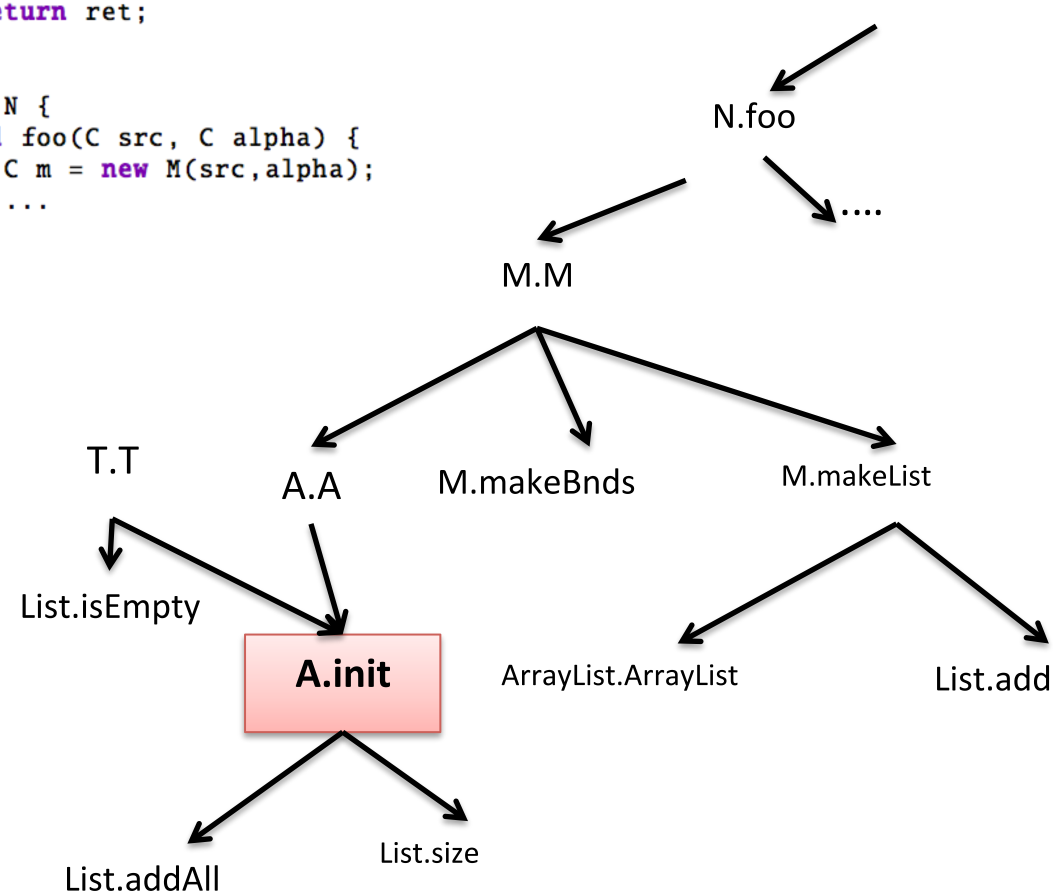
Example: NULL dereference

```
class A implements C {
  List srcs;
  A(List srcs, Rect b) {
    init (srcs, b);
  }
  void init(List srcs, Rect b) {
    this.srcs = new Vector ();
    if (srcs != null) {
      this.srcs.addAll (srcs);
    }
    if (srcs.size() != 0) {...}
  }
}
class T extends A {
  T(List srcs, Rect b) {
    if (srcs.isEmpty()) return;
    init(srcs, b);
  }
}
```

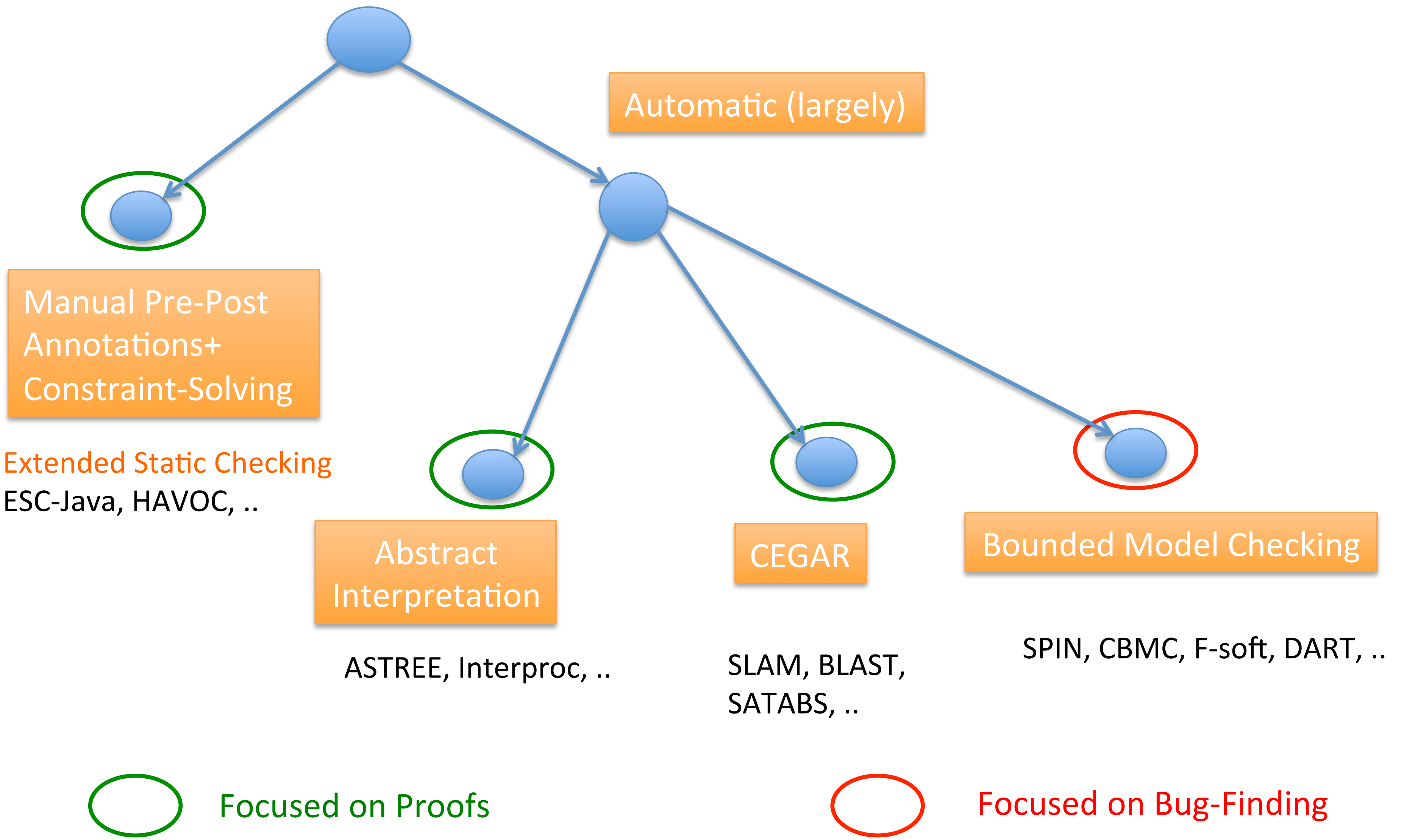
```
class M extends A {
  M(C src, C alpha) {
    List srcs; Rect b;
    srcs = makeList(src,alpha);
    b = makeBounds(src,alpha);
    super(srcs, b);
  }
  List makeList(C s1, C s2) {
    List ret = new ArrayList (2);
    ret.add(s1);
    ret.add(s2);
    return ret;
  }
}
class N {
  void foo(C src, C alpha) {
      C m = new M(src,alpha);
      ...
  }
}
```

```java
class A implements C {
  List srcs;
  A(List srcs, Rect b) {
    init (srcs, b);
  }
  void init(List srcs, Rect b) {
    this.srcs = new Vector ();
    if (srcs != null) {
      this.srcs.addAll (srcs);
    }
    if (srcs.size() != 0) {...}
  }
}
class T extends A {
  T(List srcs, Rect b) {
    if (srcs.isEmpty()) return;
    init(srcs, b);
  }
}
```

```java
class M extends A {
  M(C src, C alpha) {
    List srcs; Rect b;
    srcs = makeList(src,alpha);
    b = makeBounds(src,alpha);
    super(srcs, b);
  }
  List makeList(C s1, C s2) {
    List ret = new ArrayList (2);
    ret.add(s1);
    ret.add(s2);
    return ret;
  }
}
class N {
  void foo(C src, C alpha) {
      C m = new M(src,alpha);
      ...
  }
}
```



Main Software Verification Paradigms



Automated Verification Strategies

Hardware Verification

- Proofs on finite state space
- Iterative fact propagation towards a fixpoint (BDDs)
- Less expressive proof language (propositional logic) enables exact fixpoint computation
- Symbolic search/ simulation for finding witnesses using satisfiability solving

Abstract Interpretation

- Proofs on potentially infinite domains (integers)
- Iterative fact propagation towards a fixpoint
- Less expressive proof language
- Allow imprecision/ generalization of facts as long as the proof construction does not fail

CEGAR

- Very expressive proof language
- No generic heuristics to obtain fixpoints
- Instead, finitize the set of facts involved in proofs
- Iteratively build spurious proofs and refine them until the actual proof is obtained
- Witnesses are an after-thought

Bounded Model Checking

- Search explicitly/ symbolically on part/whole program to find bugs
- Relies on one or more “bounding assumptions”
- Proofs are an after-thought, e.g., by learning from search failures

Bounded Model Checking

- Biere, Clarke et al. ’99, originally for hardware verif.
- Initial states I, Transition relation R, Error E

  $$I(s_0) \bigwedge R_1(s_0,s_1) \bigwedge \dots \bigwedge R_n(s_{n-1},s_n) \bigwedge E(s_n)$$

  Check with a SAT/SMT solver, if satisfiable, then the solution (model) maps to an error witness

- Exploit efficient decision procedures for first-order logic, Go light on fixpoints
- Monolithic R does not scale. Need partitioned R.

Partition and Compose Iteratively

- Control/Data flow provides a natural partition

(FSM style) $R_k \cong (pc = k) \Rightarrow (x' = x + 1 \wedge pc' = k + 1)$

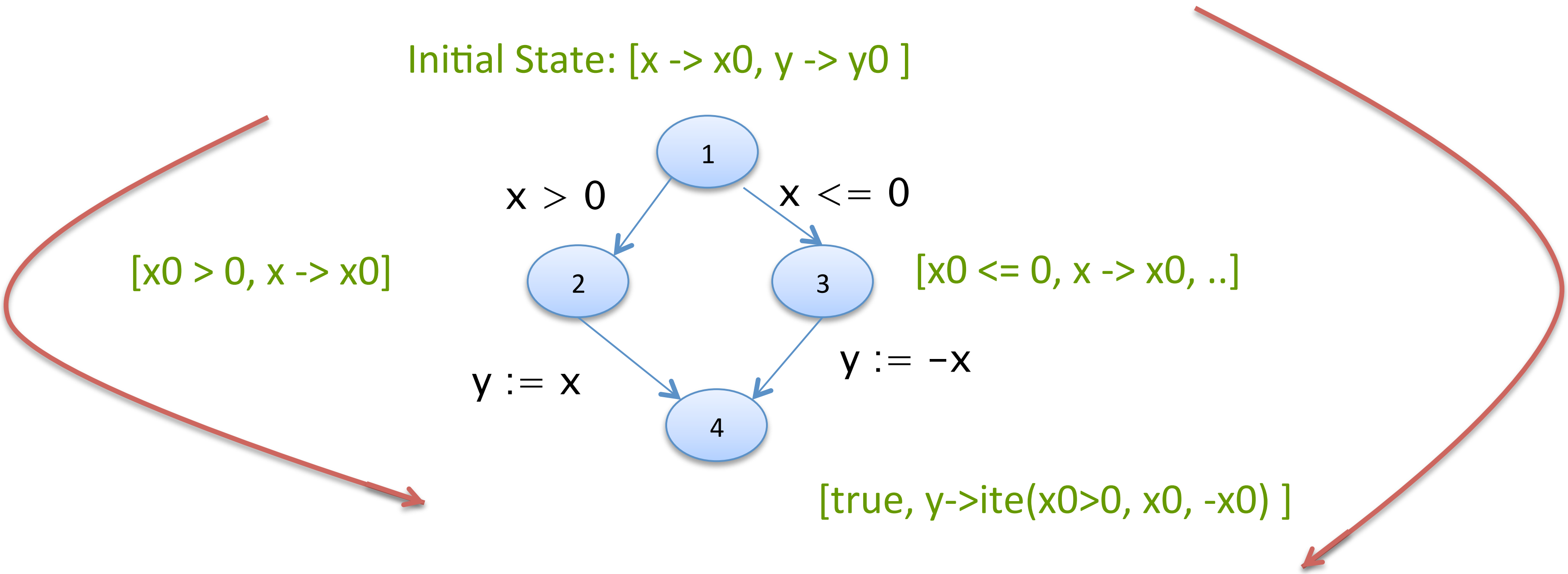
$$R = \bigwedge R_k$$

(SSA) $R_1 \cong (x_1 = x_0 + 1)$, $\quad R_2 \cong (x_3 = ite(p, x_1, x_0))$

$$R = R_1 \wedge R_2$$

- Compose R_k to simulate one or more program paths
  - Check satisfiability after composition to ensure feasibility
- These are fine-grained partitions
  – each BMC step is a small-step

Iterative Small-Step Composition



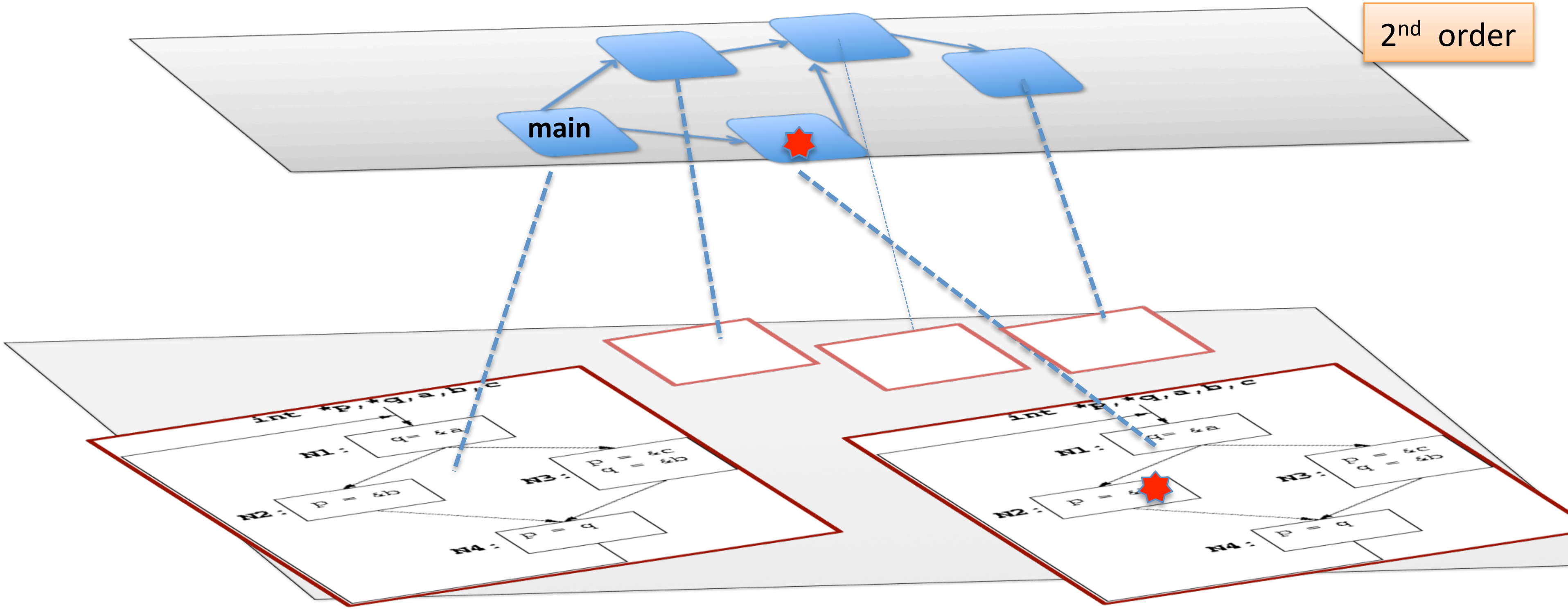
Hierarchical Program Structure

CC77c, SP81, RHS95

Call Graph

Composition of Summaries

2nd order



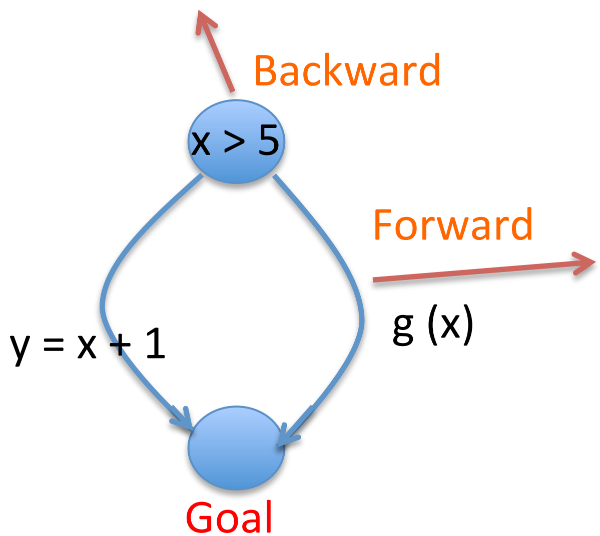
Control Flow Graphs

Computation of Summaries

1st order

Small Step composition is Bad

- If we do only small-step (or primarily small-step, reuse summaries)
  - will do repeated compositions (re-analyze procedures for different contexts)
  - hard to prioritize between composition choices



Scaling up BMC

What is the granularity of R_k ? Big-step

Which state do you start with? Goal state

How to choose between non-deterministic choice of compositions? Alternate

Alternate and Learn:
Finding Witnesses without Looking All Over

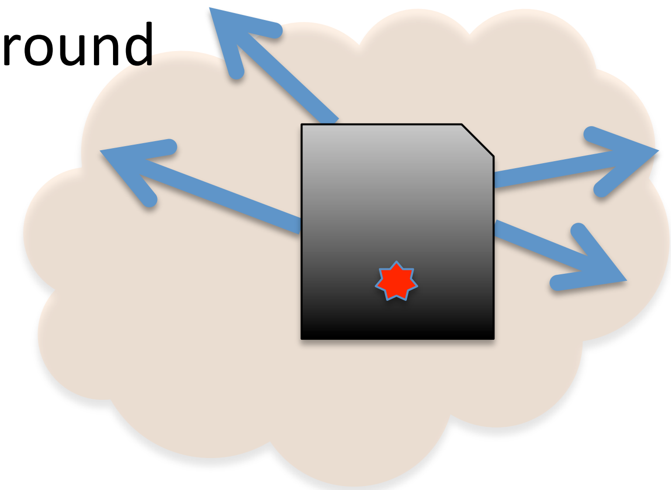
Nishant Sinha, Nimit Singhania,
Satish Chandra, Manu Sridharan

IBM Research Labs
India, U.S.A.

Computer-Aided Verification, 2012

Our approach: ALTER

- **ALTER**: Goal-Driven Big-Step Composition
  - May be viewed as Scope Expansion around a goal procedure



- Local, One-Time Summaries

- Alternating Exploration starting from Goal

- Learning from Failures

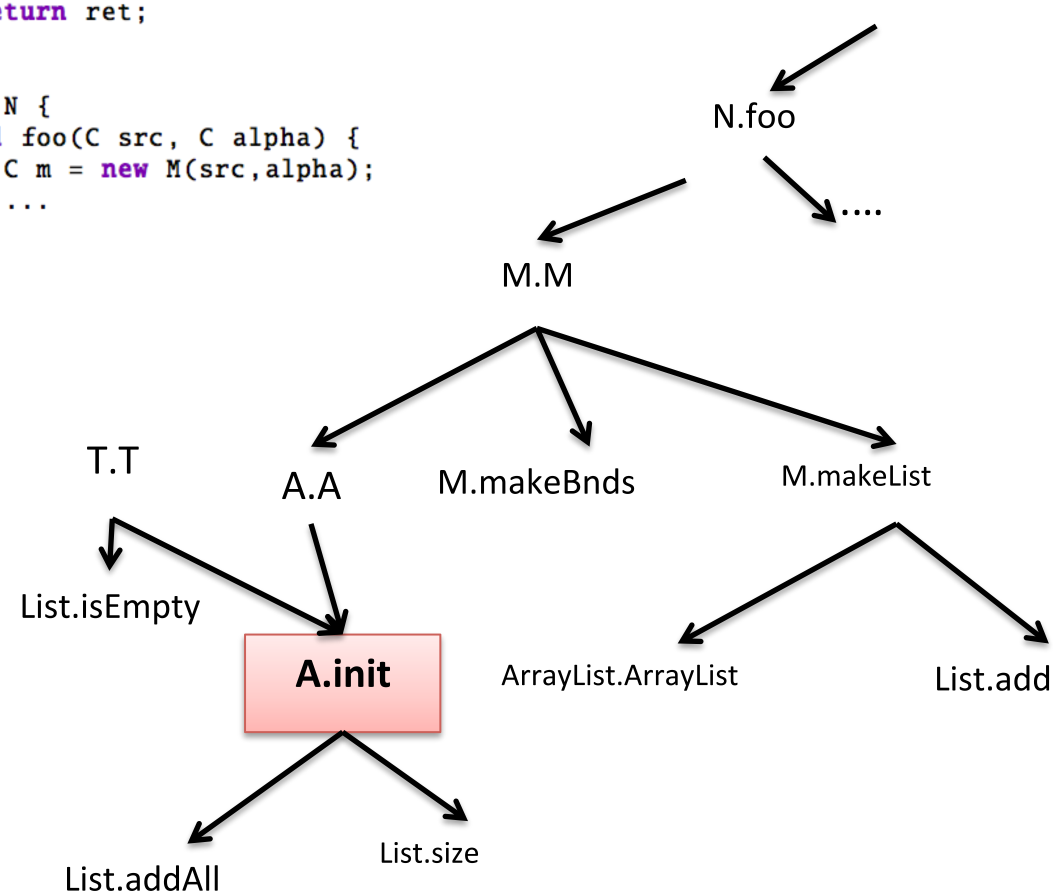
Example: NULL dereference

```java
class A implements C {
  List srcs;
  A(List srcs, Rect b) {
    init (srcs, b);
  }
  void init(List srcs, Rect b) {
    this.srcs = new Vector ();
    if (srcs != null) {
      this.srcs.addAll (srcs);
    }
    if (srcs.size() != 0) {...}
  }
}
class T extends A {
  T(List srcs, Rect b) {
    if (srcs.isEmpty()) return;
    init(srcs, b);
  }
}
```

```java
class M extends A {
  M(C src, C alpha) {
    List srcs; Rect b;
    srcs = makeList(src,alpha);
    b = makeBounds(src,alpha);
    super(srcs, b);
  }
  List makeList(C s1, C s2) {
    List ret = new ArrayList (2);
    ret.add(s1);
    ret.add(s2);
    return ret;
  }
}
class N {
  void foo(C src, C alpha) {
      C m = new M(src,alpha);
      ...
  }
}
```

```java
class A implements C {
  List srcs;
  A(List srcs, Rect b) {
    init (srcs, b);
  }
  void init(List srcs, Rect b) {
    this.srcs = new Vector ();
    if (srcs != null) {
      this.srcs.addAll (srcs);
    }
    if (srcs.size() != 0) {...}
  }
}
class T extends A {
  T(List srcs, Rect b) {
    if (srcs.isEmpty()) return;
    init(srcs, b);
  }
}
```

```java
class M extends A {
  M(C src, C alpha) {
    List srcs; Rect b;
    srcs = makeList(src,alpha);
    b = makeBounds(src,alpha);
    super(srcs, b);
  }
  List makeList(C s1, C s2) {
    List ret = new ArrayList (2);
    ret.add(s1);
    ret.add(s2);
    return ret;
  }
}
class N {
  void foo(C src, C alpha) {
      C m = new M(src,alpha);
      ...
  }
}
```



A Local Summary

<Side-effects, Call-sites, Error Conditions>

- Abstract away caller inputs and callee side-effects with fresh variables (Skolems)
- Then, compute a local summary by intra-procedural all-path symbolic execution
- Fully-precise modulo caller inputs and callee side-effects

KP05, S08, CFS09

Example: Local Summary

```
bool check (T t ) {

    if  (t != null)
          return t.f.validate();
    else
          return false;
}
```


```
bool check (T t ) {

    if  (t != null)
          return t.f.validate();
    else
          return false;
}
```


Call-Site $(t_0 \text{ != null} ,\ t \rightarrow t_0)$

Side-effect $(\text{true},\ \ \text{ret} \rightarrow \text{ite}\ (t_0 \text{ != null},\ sk_{validate} ,\ \text{false})\ )$

EC $(t_0 \text{ != null} \wedge t_0.f = \text{null})$

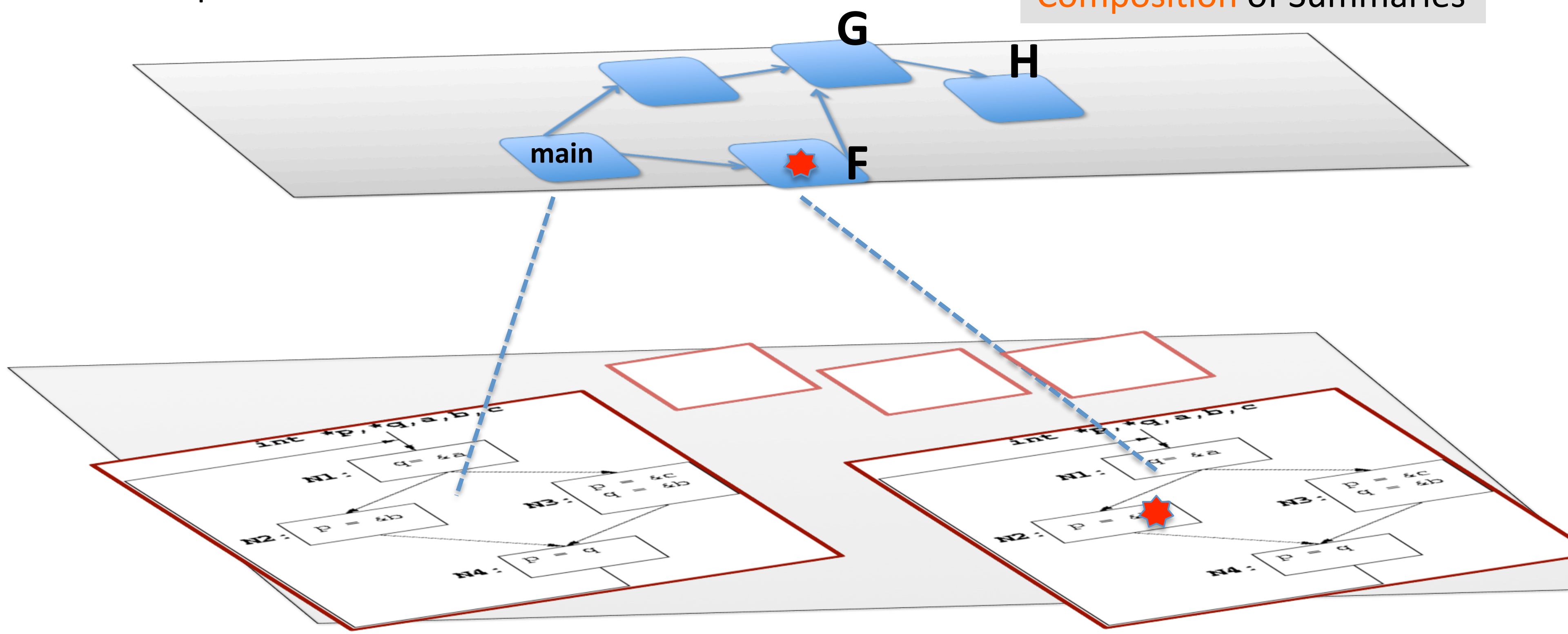
Computed only ONCE for each procedure !

Composition Strategies

CC77c, SP81, RHS95

Call Graph

Composition of Summaries



Control Flow Graphs

Computation of Summaries

1st and 2nd order composition are Intertwined

Compose with G's summary -> Compute G's summary -> Compose H -> Compute H …

Big-step Composition Strategies

- Bottom-up composition
  - Many irrelevant callees composed (e.g., large number of irrelevant virtual calls)

CC77c, SP81, RHS95

- Top-down composition
  - Many irrelevant callers composed if goal is deep

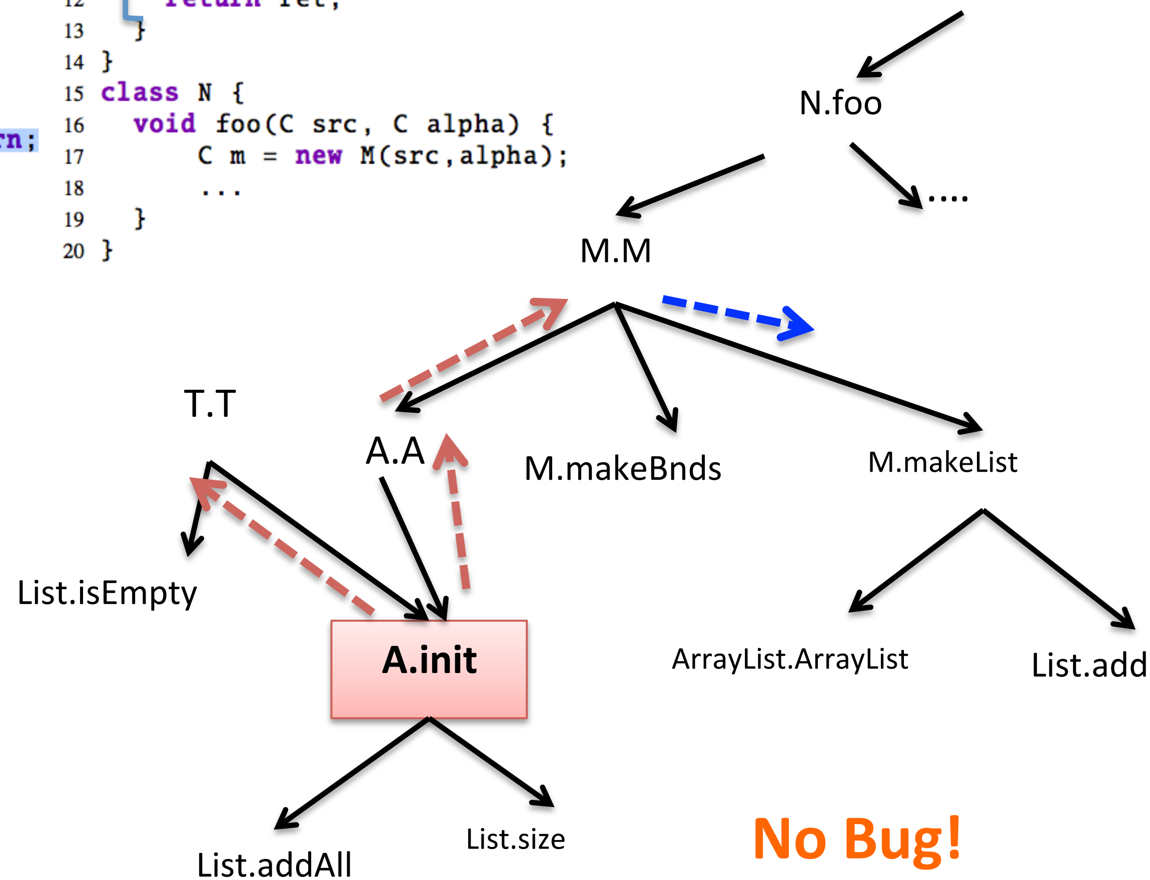
- None satisfactory, both perform eager, possibly irrelevant compositions
- Need a lazier, focused strategy

Alternating Expansion

- Start from the goal procedure
  - Pick an EC for a goal location
- Now, how do we expand out?
  - Alternate: go one caller back, then k-callees forward, further back, then fwd
  - Forward expansion (composition): Explore Callees
    - Substitute callee placeholders by side-effect summaries
  - Backward expansion: Explore Callers
    - Substitute inputs by call context summaries
- Backtrack if the composition yields no witness
- Terminate at an entrypoint

```java
class A implements C {
  List srcs;
  A(List srcs, Rect b) {
    init (srcs, b);
  }
  void init(List srcs, Rect b) {
    this.srcs = new Vector ();
    if (srcs != null) {
      this.srcs.addAll (srcs);
    }
    if (srcs.size() != 0) {...}
  }
}
class T extends A {
  T(List srcs, Rect b) {
    if (srcs.isEmpty()) return;
    init(srcs, b);
  }
}
```

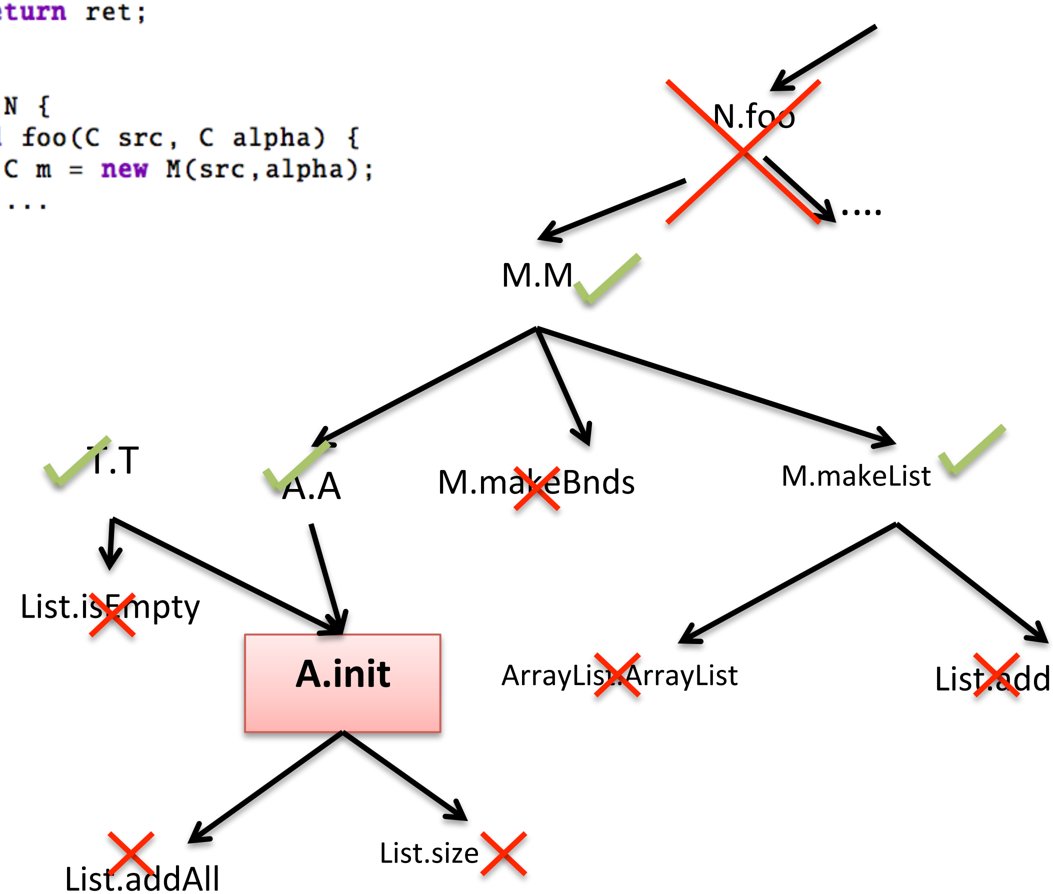
```java
class M extends A {
  M(C src, C alpha) {
    List srcs; Rect b;
    srcs = makeList(src,alpha);
    b = makeBounds(src,alpha);
    super(srcs, b);
  }
  List makeList(C s1, C s2) {
    List ret = new ArrayList (2);
    ret.add(s1);
    ret.add(s2);
    return ret;
  }
}
class N {
  void foo(C src, C alpha) {
      C m = new M(src,alpha);
      ...
  }
}
```



```java
class A implements C {
  List srcs;
  A(List srcs, Rect b) {
    init (srcs, b);
  }
  void init(List srcs, Rect b) {
    this.srcs = new Vector ();
    if (srcs != null) {
      this.srcs.addAll (srcs);
    }
    if (srcs.size() != 0) {...}
  }
}
class T extends A {
  T(List srcs, Rect b) {
    if (srcs.isEmpty()) return;
    init(srcs, b);
  }
}
```

```java
class M extends A {
  M(C src, C alpha) {
    List srcs; Rect b;
    srcs = makeList(src,alpha);
    b = makeBounds(src,alpha);
    super(srcs, b);
  }
  List makeList(C s1, C s2) {
    List ret = new ArrayList (2);
    ret.add(s1);
    ret.add(s2);
    return ret;
  }
}
class N {
  void foo(C src, C alpha) {
      C m = new M(src,alpha);
      ...
  }
}
```

Focus on program fragments relevant to a bug



Naïve Alternation is not enough

- Naïve alternation does avoid irrelevant compositions partially
  - Depends on variables in the error condition (EC)
  - But still re-visits some irrelevant callers/callees
  - Can we avoid re-exploring similar failures?

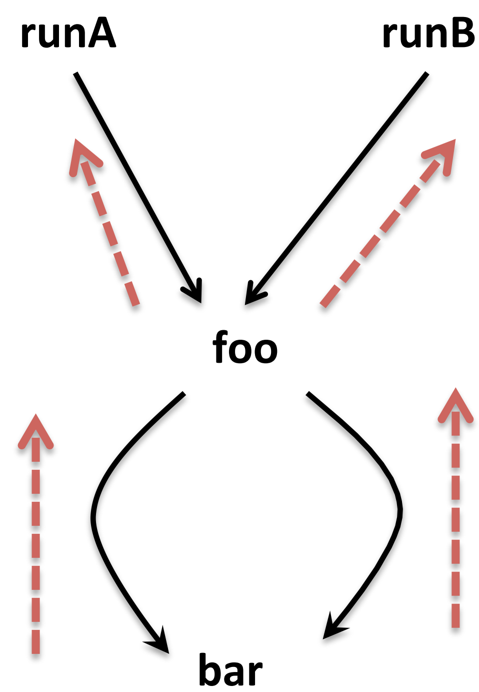
- Context explosion
  - Potentially, explore exponential number of contexts
  - Can we exploit sharing in the call graph?

Learning from Failures

- Proof-relevant procedures
  - Need to explore them at least once
  - Failure: alternating exploration into such procedures which finds no feasible path
  - On Failure, learn Invariants explaining the failure
- Learn from exploration failures
  - Learning on backtracking
  - The Invariants constitute the proof of correctness

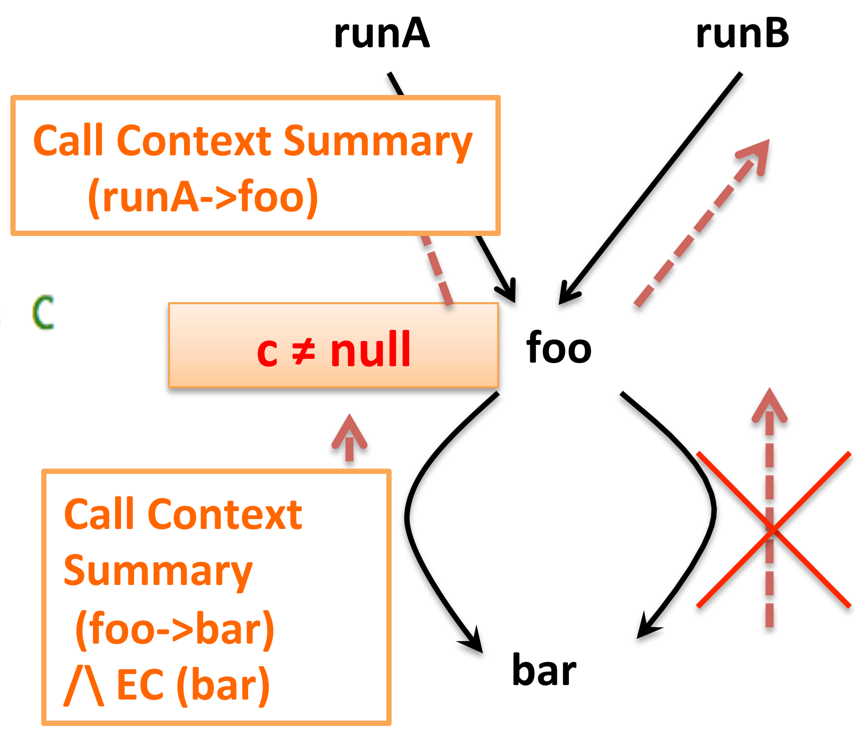
Learning from Failures

```
class App2 {
  void runA ()
    {... foo(new A()); ...)
  void runB ()
    {... foo(new B()); ...}
  //classes A, B extend class C
  int foo (C c) {
    if (*) return bar(c,1);
    else return bar(c,2);
  }
  int bar (C c, int i) {
    return c.compute(i);
  }
}
```



Learning from Failures

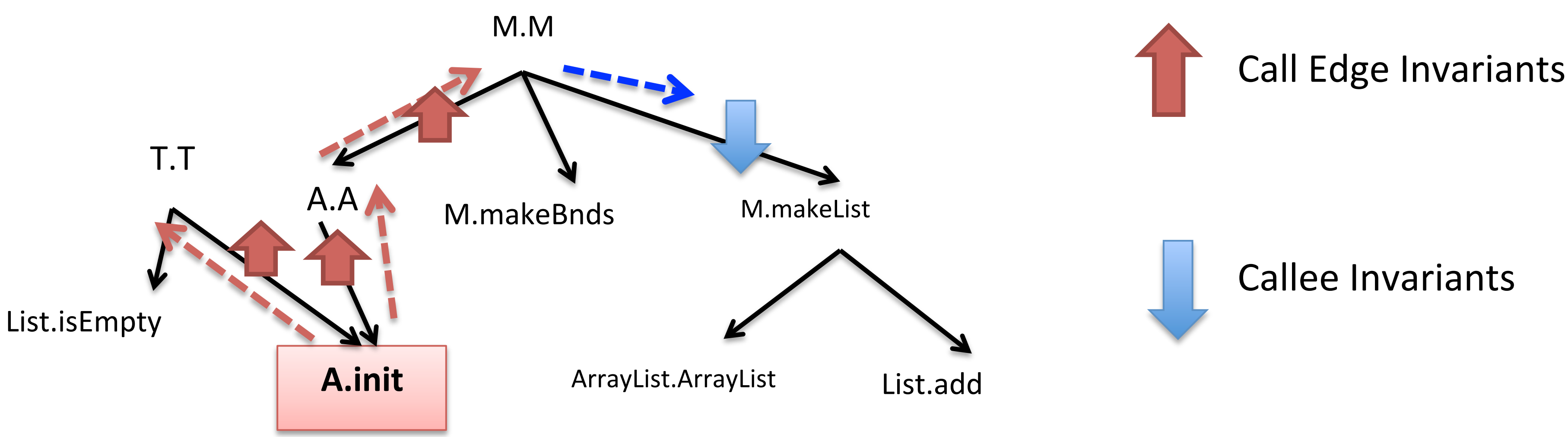
```
class App2 {
  void runA ()
    {... foo(new A()); ...)
  void runB ()
    {... foo(new B()); ...}
  //classes A, B extend class C
  int foo (C c) {
    if (*) return bar(c,1);
    else return bar(c,2);
  }
  int bar (C c, int i) {
    return c.compute(i);
  }
}
```



Interpolate!

Learning from Failures

- How do we avoid redundant composition?
  - Learn Caller and Callee invariants from failures
  - Avoid repeating similar failures (compositions) again
  - Direct towards other relevant contexts



ALTER: Implementation

- Implemented over WALA platform
  - Shares codebase with Snugglebug CFS09
  - Precise Summaries using Intra-procedural Symbolic Execution with Merging, Rewriting for Simplification
  - Incremental Call Graphs, Initial Mod-Ref
  - Triaged NULL dereference warnings from FindBugs
  - Open-source Java Benchmarks

    batik (157k), ant (88k), tomcat (163k)
- Generate and Check EC for each warning
- CVC3 for satisfiability, MathSAT5 for interpolants

Experiments: Alternation

Benchmark	WIT?	T(SB)	#FS(SB)	T(NoAlt)	MaxD(NoAlt)	#FS(NoAlt)	T(Alter)	MaxD(Alter)	#FS(Alter)
ant3	Y	>300	>154	0.6	0	1	0.6	0	1
ant4	N	4.17	102	1.9	1	2	1.21	1	2
ant5	N	2.7	66	0.8	0	1	0.87	0	1
batik2	Y	7.6	33	1.0	2	4	0.9	2	4
batik5	Y	11.5	25	18.3	23	91	5.1	9	26
batik7	N	3.5	37	> 300	> 38	> 100	1.3	3	6
batik8	N	4.5	30	2.4	1	3	2.5	1	3
batik9	Y	48.7	89	6.79	3	21	5.6	2	14
batik10	Y	3.8	88	1.7	2	4	1.8	2	4
tomcat9	N	114	26	> 300	> 16	> 74	2.8	0	7
tomcat10	Y	4.9	26	4.1	4	17	3.7	4	17
tomcat11	Y	19.64	7	0.8	0	3	0.86	0	3
tomcat12	N	> 300	>50	0.94	1	2	0.9	0	2
tomcat14	Y	6.1	26	> 300	> 17	> 55	1.778	6	7

SB = Snugglebug,
NoAlt = Explore backward till an entrypoint, then forward (no alternation)
Alter = Alternating exploration

#FS = Num Functions Summarized
MaxD = Depth of Longest Call Context explored

Experiments: Effectiveness of Learning

Benchmark	#Goals	Time(NL)	Time(L)	Time(Itp)	Edge(NL)	Edge(L)	LrnReUse	LrnEdge	LrnUpdts
ant3	9	4.013	3.996	0	8	8	0	3	3
ant4	6	1.377	1.527	0.214	3	3	0	2	3
ant5	7	1.302	1.36	0	0	0	0	0	0
batik2	20	1.349	1.589	0.183	4	4	0	1	2
batik7	23	9.319	9.529	0.546	50	41	9	6	7
batik8	24	9.113	9.179	0.461	9	9	0	2	3
batik9	32	8.508	9.879	0.931	31	31	0	8	8
batik10	20	2.558	2.45	0.306	13	9	2	2	3
tomcat9	54	24.511	26.736	0.209	68	68	0	2	2
tomcat10	33	9.193	10.542	3.203	105	39	3	23	23
tomcat11	16	2.519	2.573	0	0	0	0	0	0
tomcat12	18	4.771	4.949	0	16	16	0	0	0
tomcat14	4	2.24	1.934	0.23	17	9	4	4	4

Edge (NL/L) = num of call graph edges explored **LrnEdges**= num of edges explored with learning

LrnReuse = num of times learned invariants helped

LrnUpdt = num of times learned invariants were updated

Related

- Blast, CPAChecker
- SLAM, SMASH
- Lazy Annotations, Whale, Nested Interpolants
- Calysto, Expanding-Scope Analysis
- CBMC, FSoft DC2 platform
- Corral
- Directed Symbolic Execution

Conclusions

- Big-Step BMC for software
- Key idea: systematic, alternating exploration from the potential bug location
  - Controlled big-step compositions
  - Local summaries computed only once
  - Learn caller/callee invariants from failed explorations
- Evaluation
  - Validated bug warnings on large Java codebases
  - Alternation crucial for efficiency
  - Learning prunes exploration, but may incur cost
- Future Work
  - Better Forward expansion, Better Learning Reuse