Big-Step Bounded Model Checking for Software

Nishant Sinha,
IBM Research Labs
Bangalore, India
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) {...}
    }
}

void foo(C src, C alpha) {
    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);
        ...
    }
}
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);
        ...
    }
}
Main Software Verification Paradigms

- **Manual Pre-Post Annotations + Constraint-Solving**
  - Extended Static Checking
    - ESC-Java, HAVOC, ..

- **Abstract Interpretation**
  - ASTREE, Interproc, ..

- **Automatic (largely)**
  - CEGAR
    - SLAM, BLAST, SATABS, ..

- **Bounded Model Checking**
  - SPIN, CBMC, F-soft, DART, ..

- **Focused on Proofs**
- **Focused on Bug-Finding**
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 $E$
  \[
  I(s_0) \land R_1(s_0,s_1) \land ... \land R_n(s_{n-1},s_n) \land E(s_n)
  \]
  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
  \[ R_k \equiv (pc = k) \Rightarrow (x' = x + 1 \land pc' = k + 1) \]

\[ R = \land R_k \]

(Steeple Chase) \[ R_1 \equiv (x_1 = x_0 + 1), \quad R_2 \equiv (x_3 = \text{ite}(p, x_1, x_0)) \]

\[ R = R_1 \lor R_2 \]

• 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

Initial State: \([x \rightarrow x_0, y \rightarrow y_0]\)

- **State 1**: \(x > 0\)
  - Transition: \(y := x\)
  - Comment: \([x_0 > 0, x \rightarrow x_0]\)

- **State 2**: \(x \leq 0\)
  - Transition: \(y := -x\)
  - Comment: \([x_0 \leq 0, x \rightarrow x_0, ..]\)

- **State 3**: \(x \leq 0\)
  - Transition: \(x := x_0\)

- **State 4**: \(x > 0\)
  - Transition: \(y := \text{ite}(x_0 > 0, x_0, -x_0)\)
  - Comment: \([\text{true}, y \rightarrow \text{ite}(x_0 > 0, x_0, -x_0)]\)
Hierarchical Program Structure

Call Graph

Control Flow Graphs

Composition of Summaries

Computation of Summaries

CC77c, SP81, RHS95

1st order

2nd 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 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 T extends A {
    T(List srcs, Rect b) {
        if (srcs.isEmpty()) return;
        init(srcs, b);
    }
}

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 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 T extends A {
    T(List srcs, Rect b) {
        if (srcs.isEmpty()) return;
        init(srcs, b);
    }
}

class N {
    void foo(C src, C alpha) {
        C m = new M(src, alpha);
        ...
    }
}
```
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
Example: Local Summary

```c
bool check (T t ) {
    if (t != null)
        return t.f.validate();
    else
        return false;
}
```

Call-Site

(t₀ != null, t → t₀)

Side-effect

(true, ret → ite (t₀ != null, sk_validate, false))

EC

(t₀ != null ∧ t₀.f = null)

Computed only ONCE for each procedure!
Composition Strategies

1\textsuperscript{st} and 2\textsuperscript{nd} order composition are Intertwined
Compose with G’s summary \rightarrow Compute G’s summary \rightarrow Compose H \rightarrow Compute H ...
Big-step Composition Strategies

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

- **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
No Bug!
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
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
  – 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

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

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

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

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

**LrnReuse** = num of times learned invariants helped  
**LrnUpdts** = num of times learned invariants were updated
• 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