Model Checking with Abstract State Matching

Corina Păsăreanu
_QSS, NASA Ames Research Center_

Joint work with
Saswat Anand (Georgia Institute of Technology)
Radek Pelánek (Masaryk University)
Willem Visser (RIACS, NASA Ames)
Introduction

• Abstraction in software model checking
  – Used to reduce data domains of a program
  – Described as abstract interpretation
  – Classic approach: over-approximation
    • SLAM, Blast, Magic; see also Bandera, Feaver
  – Preserves true results; abstract counter-examples may be infeasible
  – Counter-example based iterative abstraction refinement

Our approach
• Under-approximation based abstraction with refinement
  – Goal: error detection; explores only feasible system behaviors
  – Preserves errors of safety properties
  – Iterative refinement based on checking “exactness” of abstraction
• Framework for test input generation – built around Java PathFinder
  – Measure code coverage
  – Evaluate against other test input generation methods
  – Applied to Java container classes
Predicate Abstraction

- Maps a (possibly infinite state) concrete transition system into a finite state system
  - Via a set of predicates: $\text{Preds} = \{p_1, p_2 \ldots p_n\}$
- Abstraction function $\alpha$: $\text{ConcreteStates} \rightarrow \text{BitVectors}$
  \[ \alpha(s) = b_1b_2...b_n, \quad b_i=1 \iff s \models p_i \]

Traditional approaches:
- **May** abstract transitions:
  - Over-approximate concrete transitions
  - $a_1 \xrightarrow{\text{may}} a_2 : \exists s_1 \text{ s.t. } \alpha(s_1)=a_1 \text{ and } \exists s_2 \text{ s.t. } \alpha(s_2)=a_2, \text{ s.t. } s_1 \rightarrow s_2$
- **Must** abstract transitions:
  - Under-approximate concrete transitions
  - $a_1 \xrightarrow{\text{must}} a_2 : \forall s_1 \text{ s.t. } \alpha(s_1)=a_1, \exists s_2 \text{ s.t. } \alpha(s_2)=a_2 \text{ and } s_1 \rightarrow s_2$

- Compute may/must transitions automatically:
  - Use a theorem prover/decision procedure: require $2^n \times n \times 2$ calls
Concrete search with abstract matching:

• Traverse the concrete system
• For each explored concrete state
  – Store abstract version of the state
  – Use predicate abstraction
• Abstract state used to determine if the search should continue or backtrack

• Does not build abstract transitions
  – It executes the concrete transitions directly

• Decision procedure invoked during refinement:
  – At most 2 calls for each explored transition
Example

Concrete system

May abstraction
\[ p = (x < 2) \]
Over-approximation

Must abstraction
\[ p = (x < 2) \]
Under-approximation
Example

<table>
<thead>
<tr>
<th>Concrete system</th>
<th>May abstraction</th>
<th>Must abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = (x &lt; 2)</td>
<td>Over-approximation</td>
<td>Under-approximation</td>
</tr>
</tbody>
</table>

Diagram:

```
abstraction α

A,0 → B,1 → D,1 → E,2
   ↑        ↑
   ↓        ↓
A,p → B,p → D,p → E,p
   ↑        ↑
   ↓        ↓
C,0 → E,1 → E,0
   ↑
   ↓
B,p → C,p
```

- **Concrete system**
  - A,0
  - B,1
  - C,0
  - D,1
  - D,0
  - E,2
  - E,1

- **May abstraction**
  - p = (x < 2)
  - Over-approximation

- **Must abstraction**
  - p = (x < 2)
  - Under-approximation
Example

Concrete system

May abstraction
\( p = (x < 2) \)
Over-approximation

Must abstraction
\( p = (x < 2) \)
Under-approximation
Example

Concrete system

May abstraction
\( p = (x < 2) \)
Over-approximation

Must abstraction
\( p = (x < 2) \)
Under-approximation

Concrete search w/ abstract matching
\( p = (x < 2) \)
Example

Concrete system

- A,0
  - B,1
    - D,1
      - E,2
  - C,0
    - D,0
      - E,1

May abstraction
- p = (x < 2)
  - Over-approximation

- A, p
  - B, p
    - D, p
      - E, !p
  - C, p

Must abstraction
- p = (x < 2)
  - Under-approximation

- A, p
  - B, p
    - D, p

Concrete search with abstract matching
- p = (x < 2)
Example

Concrete system
May abstraction
\( p = (x < 2) \)
Over-approximation

Must abstraction
\( p = (x < 2) \)
Under-approximation

Concrete search w/ abstract matching
\( p = (x < 2) \)
Example

Concrete system

May abstraction
\( p = (x < 2) \)
Over-approximation

Must abstraction
\( p = (x < 2) \)
Under-approximation

Concrete search w/ abstract matching
\( p = (x < 2) \)

Concrete search w/ abstract matching
\( p = (x < 2) \)
\( q = (x < 1) \)
PROCEDURE dfs()

BEGIN
    add(s₀, VisitedStates);
push(s₀, Stack);
WHILE ! empty(Stack) DO
    s = pop(Stack);
    FOR all transitions t enabled in s DO
        s' = successor(s, t);
        IF s' NOT IN VisitedStates THEN
            add(s', VisitedStates);
push(s', Stack);
        FI;
    OD;
END;
PROCEDURE \( \alpha \text{Search} (\text{Preds}) \)

BEGIN

add(\( \alpha_{\text{Preds}} (s_0) \), VisitedStates);
push(s_0, Stack);
WHILE ! empty(Stack) DO
  s = pop(Stack);
  FOR all transitions t enabled in s DO
    s' = successor(s, t);
    IF \( \alpha_{\text{Preds}} (s') \) NOT IN VisitedStates THEN
      add(\( \alpha_{\text{Preds}} (s') \), VisitedStates);
push(s', Stack);
    FI;
  OD;
OD;
END;
Abstraction Refinement

Check if abstraction is **exact** with respect to each transition $t: s \rightarrow s'$
- Check if the induced abstract transition is a **must** transition w/ a decision procedure
- If not, add new predicates
- Use weakest precondition calculations $\alpha(s) \Rightarrow \wp(\alpha(s'),t)$

Abstraction is exact

Abstraction is refined

Add **new predicate** $(x-1>0)$ from failed check and repeat $\alpha$Search
Iterative Refinement

- Check if bad state $\varphi_{err}$ is reachable

BEGIN
  Preds = $\emptyset$;
  WHILE true DO
    $\alpha$Search(Preds);
    /* during $\alpha$Search perform:
    - IF $\varphi_{err}$ is reachable THEN output counterexample FI;
    - check if abstraction is exact for each transition
    - NewPreds = newly generated predicates from failed checks
    */
    IF NewPreds = $\emptyset$ THEN output unreachable FI;
    Preds = Preds $\cup$ NewPreds;
  OD;
END;
Correctness and Termination

- In general
  - The iterative algorithm might not terminate
- If it terminates
  - It finds an error or
  - It computes a finite bisimilar structure
- If a finite (reachable) bisimulation quotient exists then
  - It will eventually compute a finite bisimilar structure
  - May still fail to terminate
Implementation

• Implementation for simple guarded command language
  – PERL, OCAML
  – Uses SIMPLIFY as a decision procedure

Applications

• Property verification for the Bakery mutual exclusion protocol
  – Search order matters
    5 iterations for breadth first search order
    4 iterations for depth first search order

• Error detection in RAX (Remote Agent Executive)
  – Component extracted from an embedded spacecraft-control application
  – Deadlocked in space
  – Error found faster than over-approximation based analysis
Related Work

- Refinement of under-approximations
  - For SAT based bounded model checking – Grumberg et al. [POPL’05]
- May and must abstractions
  - Branching time properties – Godefroid et al [Concur’01]
  - “Hyper” must transitions for monotonicity – Shoham and Grumberg [TACAS’04]
  - Dams and Namjoshi, de Alfaro et al [LICS’04], Ball et al [CAV’05]
  - Our previous work – choice free search [TACAS’01]
- Model driven software verification
  - Use abstraction mappings during concrete model checking – Holzmann and Joshi [SPIN’04]
- Over-approximation based predicate abstraction
- Online minimization of transition systems
  - Lee & Yannakakis [1992]
Conclusions (I)

- Model checking algorithm
  - Under-approximation refinement
  - Integrates abstract analysis with concrete program execution
  - Uses decision procedure to detect incompleteness of abstraction and to refine the abstraction

- Comparison with standard over-approximation abstraction
  - Finds errors faster (potentially)
  - More efficient (in the number of theorem prover calls)
  - Complementary, should be combined

- Future work
  - Liveness properties
  - Backward vs. forward refinement, property driven refinement
  - Evaluation
Part II
Test Input Generation

- Model checking with abstract state matching
  - No automated refinement
  - User-provided abstractions
- Generate test input sequences for Java container classes
  - Use Java PathFinder (JPF)
    - Explicit state model checker for Java programs
  - (Abstract) state matching
    - To avoid generation of redundant test sequences
  - Measure coverage
    - Whenever coverage increased, output test sequence
- Test oracles
  - Method post-conditions, assertions
  - Absence of run-time errors
Test sequence: add(1); add(0); remove(0);
Driver Skeleton

M: sequence length
N: parameter values

Container c = new Container();
for (int i = 0; i < M; i++) {
    int v = Verify.random(N - 1);
    switch (Verify.random(1)) {
        case 0: c.add(v); break;
        case 1: c.remove(v); break;
    }
    Verify.ignoreIf(checkAbstractState(c));
}
Test Generation Techniques

• Explicit state model checking
  – “Classical” concrete state matching
  – Abstract state matching

• Model checking with symbolic execution
  – State matching using subsumption checking
  – Abstract matching

• Model checking with random selection
Abstract Matching

• Perform state matching after each method call
  – Map container state to an abstract version
  – Backtrack if abstract state was seen before, i.e. discard test sequence

• Automated support for two abstractions:
  – Shape abstraction
    • Records (concrete) heap shape of container; discards numeric data
    • Obtained through heap “linearization”
    • Comparing shapes reduces to comparing sequences
  – “Complete” abstraction
    • Shape augmented with data
    • Similar to symmetry reduction in software model checking
Complete Abstraction: Shape + Data

17 23 31 0 0 45 0 0 58 0 0

≠

16 23 32 0 0 45 0 0 58 0 0
Symbolic Execution

- Execute methods on symbolic input values
- Symbolic states represent **sets** of concrete states
  - Can yield significant improvement over explicit execution
- For each path, build a **path condition**
  - Condition on inputs – for the execution to follow that path
  - Check satisfiability
Example – Explicit Execution

Code that swaps 2 integers:

```c
int x, y;
if (x > y) {
    x = x + y;
y = x - y;
x = x - y;
if (x > y)
    assert false;
}
```

Concrete Execution Path:

- `x = 1, y = 0`
- `1 > 0 ? true`
- `x = 1 + 0 = 1`
- `y = 1 - 0 = 1`
- `x = 1 - 1 = 0`
- `0 > 1 ? false`
Example – Symbolic Execution

Code that swaps 2 integers:

```c
int x, y;
if (x > y) {
    x = x + y;
    y = x - y;
    x = x - y;
    if (x > y)
        assert false;
}
```

Symbolic Execution Tree:

```
path condition
[PC:true] x = X, y = Y

[PC:true] x = X+Y
true
[PC:X>Y] y = X + Y – Y = X
false
[PC:X>Y] x = X + Y – X = Y
true
[PC:X>Y] Y > X ?
false
[PC:X>Y ∧ Y≤X ] END
true
[PC:X>Y ∧ Y>X] END
```

Example – Symbolic Execution
Symbolic Execution in JPF

• Handles dynamically allocated data, arrays, concurrency
• Uses Omega library for linear integer constraints
• State matching
  – Subsumption between symbolic states
Symbolic State

Shape

Symbolic Constraints

\[ e_1 > e_2 \wedge e_2 > e_3 \wedge e_2 < e_4 \wedge e_5 > e_1 \]
Subsumption Checking

Stored state:

```
\[
\begin{align*}
& e_1 > e_2 \land \\
& e_2 > e_3 \land \\
& e_2 < e_4 \land \\
& e_5 \geq e_1
\end{align*}
\]
```

Same shape \[\uparrow\] Matched

New state:

```
\[
\begin{align*}
& e_1 > e_2 \land \\
& e_2 > e_3 \land \\
& e_2 < e_4 \land \\
& e_5 > e_1
\end{align*}
\]
```

Set of concrete states represented by stored state

Set of concrete states represented by new state
Subsumption Checking

Existential Quantifier Elimination

\( e_1 = V_1 \land e_2 = V_4 \land e_3 = V_3 \land e_4 = V_5 \land e_5 = V_2 \)

PC:
\( V_1 < V_2 \land V_4 > V_3 \land V_4 < V_1 \land V_4 < V_5 \land V_7 < V_2 \land V_7 > V_1 \)

\( \exists V_1, V_2, V_3, V_4, V_5, V_7: \\
  e_1 = V_1 \land e_2 = V_4 \land e_3 = V_3 \land e_4 = V_5 \land e_5 = V_2 \land PC \\
  \) simplifies to
\( e_1 > e_2 \land e_2 > e_3 \land e_2 < e_4 \land e_5 > e_1 \)
Evaluation

- Four container classes
  - BinaryTree, BinomialHeap, FibonacciHeap, TreeMap
- Measured coverage
  - Number of basic blocks covered by the generated tests
- Measured predicate coverage – at each basic block
  - Combinations of predicates chosen from conditions in the code
  - More difficult to achieve

- Breadth first search order
- Sequence Length = Number of Values (M=N)
  - Tried other values
- Dell Pentium 4, 2.2 GHz, Windows 2000, 1GB memory
- Out of Memory runs not considered
TreeMap – Basic Block Coverage

### Exhaustive Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Coverage</th>
<th>Seq Length</th>
<th>Time (s)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Checking</td>
<td>37</td>
<td>6</td>
<td>38</td>
<td>243</td>
</tr>
<tr>
<td>Complete Abstraction</td>
<td>39</td>
<td>7</td>
<td>9</td>
<td>34</td>
</tr>
<tr>
<td>SymEx w/ Subsumption</td>
<td>39</td>
<td>7</td>
<td>15</td>
<td>22</td>
</tr>
</tbody>
</table>

### Lossy Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Coverage</th>
<th>Seq Length</th>
<th>Time (s)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape Abstraction</td>
<td>39</td>
<td>10</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>SymEx w/ Shape Abstraction</td>
<td>39</td>
<td>7</td>
<td>7</td>
<td>22</td>
</tr>
<tr>
<td>Random Selection</td>
<td>39</td>
<td>10</td>
<td>18</td>
<td>5</td>
</tr>
</tbody>
</table>
# TreeMap – Predicate Coverage

## Exhaustive Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Coverage</th>
<th>Seq Length</th>
<th>Time (s)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model Checking</td>
<td>55</td>
<td>6</td>
<td>38</td>
<td>229</td>
</tr>
<tr>
<td>Complete Abstraction</td>
<td>95</td>
<td>10</td>
<td>271</td>
<td>844</td>
</tr>
<tr>
<td>SymEx w/ Subsumption</td>
<td>104</td>
<td>12</td>
<td>594</td>
<td>896</td>
</tr>
</tbody>
</table>

## Lossy Techniques

<table>
<thead>
<tr>
<th>Technique</th>
<th>Coverage</th>
<th>Seq Length</th>
<th>Time (s)</th>
<th>Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shape Abstraction</td>
<td>106</td>
<td>20</td>
<td>281</td>
<td>1016</td>
</tr>
<tr>
<td>SymEx w/ Shape Abstraction</td>
<td>102</td>
<td>13</td>
<td>1309</td>
<td>1016</td>
</tr>
<tr>
<td>Random Selection</td>
<td>106</td>
<td>39</td>
<td>78</td>
<td>17</td>
</tr>
</tbody>
</table>
Observations

Coverage
- Basic block coverage – easily achieved with all techniques
- Predicate coverage
  - Difficult to achieve with “classical” model checking
  - Its close “cousin” (complete abstraction) scales better
  - Lossy techniques better than exhaustive ones

Symbolic vs. explicit execution
- Exhaustive – subsumption checking
  - Better than exhaustive concrete execution
- Lossy – abstract matching
  - Worse than concrete search with abstract matching

Random selection
- Requires longer sequences to achieve good coverage
- Could not obtain “best” coverage for FibonacciHeap and BinomialHeap (more interface methods with more parameters)
  - Concrete search with abstract matching performed better
Conclusions (II)

• Test input generation techniques for Java containers
  – State matching to avoid generation of redundant tests
  – Concrete/abstract matching, explicit/symbolic execution
  – Compared to random selection

• Model checking with shape abstraction
  – Good coverage with short sequences
  – Shape abstraction provides an accurate representation of containers

• Future work
  – Coverage highly dependent on abstraction – automatic refinement
  – Complex data structures, arrays as input parameters
  – Abstractions used in shape analysis [SPIN’06]
  – More experiments
  – Measure techniques in terms of defect detection, rather than coverage
Explanation

- **Bisimulation:** symmetric relation $\sim$
  - $s \sim s'$ iff for every $s \rightarrow s_1$ there exists $s' \rightarrow s_1'$ s.t. $s_1 \sim s_1'$

- Two transition systems are bisimilar if
  - Their initial states are bisimilar

- $\sim$ induces a **quotient** transition system
  - States are equivalence classes
  - $A \rightarrow B$ if there exist $s$ in $A$ and $s'$ in $B$ s.t. $s \rightarrow s'$
Non-monotonic Refinement

State space explored

Refinement

Matched