Abstraction

Pavel Parízek
Motivating example

1: int sum(int from, int to) {
2:     int total = 0;
3:     for (int i = from; i <= to; i++) {
4:         total += i;
5:     }
6:     return total;
7: }

8: main() {
9:     int x = sum(1, 1000);
10:    assert(x > 0);
11: }
Abstraction

• Goal: smaller reachable program state space

• Approaches
  • Reducing the size of variables’ data domains
  • Ignoring concrete values of certain variables

• Benefits
  • Mitigating the state space explosion
  • Improved scalability (performance)
Data abstraction

- Using abstract domains for program variables
- Tracking only abstract states of the program

- Abstract state = set of concrete states

- Process: mapping **concrete** to **abstract**
  - data types, values, operations, program states
Example: Signs abstraction

- Abstract data type
  - int $\mapsto$ \{ NEG, ZERO, POS \}

Q: What about values and operations? Let’s consider only addition here.
Example: Signs abstraction

- Abstract data type
  - \texttt{int} \xrightarrow{} \{ \text{NEG, ZERO, POS} \}

- Abstract values
  - \( \alpha(x) \subseteq \{ \text{NEG, ZERO, POS} \} \)

- Abstract operation +

<table>
<thead>
<tr>
<th></th>
<th>NEG</th>
<th>ZERO</th>
<th>POS</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEG</td>
<td>{ NEG }</td>
<td>{ NEG }</td>
<td>{ NEG, ZERO, POS }</td>
</tr>
<tr>
<td>ZERO</td>
<td>{ NEG }</td>
<td>{ ZERO }</td>
<td>{ POS }</td>
</tr>
<tr>
<td>POS</td>
<td>{ NEG, ZERO, POS }</td>
<td>{ POS }</td>
<td>{ POS }</td>
</tr>
</tbody>
</table>
Construction of abstract programs

- Transformation of program source code

```java
int x = 0;
...
int y;
y = x + 2;
```

```java
Signs x = Signs.ZERO;
...
Signs y;
y = Signs.add(x, Signs.POS);
```
Abstract state space

- Non-deterministic choice
  - assignment, branching condition (if-else, loops)

```c
int x = 5;
int y = -2;
z = x + y;
```

```
x: POS
y: NEG
```

```
z = x + y
```

```
z: {NEG, ZERO, POS}
z: NEG
z: ZERO
z: POS
```
Other simple data abstractions

- Interval abstraction
  - Example: \( x < 0, \ 0 \leq x \leq 10, \ x > 10 \)

- Combining intervals with concrete values
  - Example: \( x < 0, \ x = 0, \ x = 1, \ x = 2, \ x = 3, \ x = 4, \ x > 4 \)
Predicate abstraction
Predicate abstraction

- Data type
  - Predicates about program variables
    - Theories: linear integer arithmetic, equality, arrays
    - Example: $x = 0$, $x > 0$, $y + z \geq 2$, $u = v$, $select(a,1) = 5$

- Abstract state
  - Some valuation of all the predicates
1: int sum(int from, int to) {
2:     int total = 0;
3:     for (int i = from; i <= to; i++) {
4:         total += i;
5:     }
6:     return total;
7: }

8: int x = sum(1, 1000);
9: assert(x > 0);

Q: what predicates should we use here?
bool P1 = false;
bool P2 = false;

/* int total = 0;
P2 = true;

/* int i = from;
P1 = *;

/* total += i;
if (P1 && P2) P2 = true;
else P2 = *;

Predicates
P1: i > 0
P2: total ≥ 0
Deriving predicate valuations

- Weakest preconditions
  - Predicate \( p \): total \( \geq 0 \)
  - Statement \( s \): total += \( i \);
  - WP(\( s, p \)) \( \equiv \) total + \( i \) \( \geq 0 \)

- Querying the SMT solver
  - Example: \( p1 \) && \!p2 \( \rightarrow \) WP(\( s, p \)) is valid ?

- Processing results
  1) \( p1 \) && \!p2 \( \rightarrow \) WP(\( s, p \)) is valid \( \rightarrow \) if (\( p1 \) && \!p2) \( p = \) true;
  2) \( p1 \) && \!p2 \( \rightarrow \) WP(\( s, !p \)) is valid \( \rightarrow \) if (\( p1 \) && \!p2) \( p = \) false;
  3) both valid or none valid \( \rightarrow \) if (\( p1 \) && \!p2) \( p = *; \)
Optimizations

- Goal: reduce the number of queries for SMT

- Possible approaches
  - Compute new valuation only for predicates that refer to variables modified by the given concrete assignment statement
    - We must be very careful though: aliasing
  - For generating branches of the big if-else statements in the abstract boolean program, consider only predicates that refer to variables read by the assignment statement
Verification using predicate abstraction

- Using model checker for boolean programs
  - Much easier task than for general programs (C, Java)
  - Well-known optimizations: symbolic model checking

- Practical challenges
  - Translating counterexamples back to source code
  - Encoding properties into reachability of assertions
Assume that we want to verify a given program.

Q: What important characteristic should the abstract program have?
Over-approximation

- Abstract program captures all possible behaviors of the original concrete program
  - Behavior: possible control flow path, thread interleaving

- Purpose: complete verification (all reachable states)

- Examples
  - Simple data abstraction
  - Predicate abstraction

- Problem: imprecise abstraction
  - Captures some infeasible execution paths \(\rightarrow\) spurious errors
  - Branch conditions replaced with a non-deterministic choice
Q1: Is there some other way to creating abstract programs than over-approximation?

Q2: If yes, when does it make sense to use it?
Under-approximation

- Abstract program captures only a certain subset of all possible behaviors of the concrete program
  - selected thread interleavings, reduced data domains

- Purpose: fast error detection (subset of reachable states)

- Examples
  - State space traversal with heuristics
  - Context-bounded search (traversal)
  - Bounded model checking in general

- Problem: imprecise abstraction
  - Omits some feasible execution paths ➔ missed errors
Abstractions: characteristics

<table>
<thead>
<tr>
<th>Over-approximation</th>
<th>Under-approximation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error in abstraction</td>
<td>Error in abstraction</td>
</tr>
<tr>
<td>Error in concrete program</td>
<td>Error in concrete program</td>
</tr>
<tr>
<td>Error-free abstraction</td>
<td>Error-free abstraction</td>
</tr>
<tr>
<td>Error-free concrete program</td>
<td>Error-free concrete program</td>
</tr>
<tr>
<td><strong>Spurious errors</strong></td>
<td><strong>Missed errors</strong></td>
</tr>
</tbody>
</table>
Abstraction: issues

• Very hard to get right
  ▪ Too precise ➔ state explosion
  ▪ Too coarse ➔ spurious errors

• Possible remedy
  ▪ Start with coarse abstraction
  ▪ Employ iterative refinement
Counter-Example Guided Abstraction Refinement

- Automated iterative refinement based on spurious errors
More predicates

Refinement

Predicates

Abstraction

Concrete program

Boolean program

BP model checker

Error trace analysis

feasible

“Real bug found”

ERROR

Error trace

OK

“System is safe”

Picture created by Ondřej Šerý
Challenges

- Checking error trace feasibility
- Inferring additional predicates
Checking error trace feasibility

- Record the path condition \textit{PaC} using symbolic execution
  - Options selected at choice points (if-else, loops, non-determinism)

- Create path formula that encodes the whole error trace
  - The \texttt{assume} statement: clauses from the \textit{PaC} (selected branches)

- Check satisfiability of the path formula (query the SMT solver)

- Example
  - Error trace
    \[
    \text{index} = 1; \text{total} = \text{total} + \text{index}; \text{assume} \text{index} > 1000
    \]
  - Path formula
    \[
    (\text{index0} = 1) \land (\text{total1} = \text{total0} + \text{index0}) \land (\text{index0} > 1000)
    \]
Inferring additional predicates

- Divide path formula $\phi$ into two parts $\phi^-$ and $\phi^+$
  - such that $\phi^- \land \phi^+$ is unsatisfiable
- Then derive a Craig interpolant $\psi$ for $\phi^-$ and $\phi^+$
  - Logic formula $\psi$ such that
    - $\phi^- \rightarrow \psi$, $\phi^+ \land \psi$ is unsatisfiable, and
    - $\psi$ uses symbols common to $\phi^-$ and $\phi^+$
- Finally generate additional predicates from $\psi$

Example

- Path formula
  - $(\text{index0} = 1) \land (\text{total1} = \text{total0} + \text{index0}) \land (\text{index0} > 1000)$
  - $\phi^-: \text{index0} = 1 \land \text{total1} = \text{total0} + \text{index0}$
  - $\phi^+: \text{index0} > 1000$
  - $\psi: \text{index0} = 1$ // newly inferred predicate in this case

Disclaimer

- Bad choices of inferred predicates may lead to non-termination
- Tools generate predicates that may look strange (not intuitive)
Static Driver Verifier (SDV)
- SLAM: verification engine that uses CEGAR

Purpose
- Analyzing third party Windows device drivers
  - Specific rules about proper usage of Windows kernel API
  - Major source of kernel crashes (infamous “blue screens”)
  - Drivers have feasible code size and a strict environment

Many extensions developed in the last decade

Additional information
- Many research papers, slides, download, user guides
Optimizations

• Lazy abstraction
  ▪ Set of predicates specific to each code location
  ▪ Tools: BLAST

• Method summaries
  ▪ Logic formula relating inputs and outputs
  ▪ Summaries computed using interpolants
  ▪ Tools: Whale, FunFrog, ...
Tools

- BLAST
  - http://mtc.epfl.ch/software-tools/blast/
- CPAchecker
  - http://cpachecker.sosy-lab.org/
- UFO/Whale
  - https://bitbucket.org/arieg/ufo
- Wolverine
- ... and many others
Further reading


