Compositional Program Analysis & Invariant Generation using Program Abstraction

Jakub Šárník

ParaDiSe seminar, Spring 2021

Contents

Motivation

- We want: program analysis with nondeterministic data
- We saw: abstraction support in DIVINE with abstract and symbolic representations

Motivation

- We want: program analysis with nondeterministic data
- We saw: abstraction support in DIVINE with abstract and symbolic representations
- We have: problems
  - intractable to analyse whole programs at once (i.e. a single verification run from the entry point to all reachable states)

Motivation

- We want: program analysis with nondeterministic data
- We saw: abstraction support in DIVINE with abstract and symbolic representations
- We have: problems
  - intractable to analyse whole programs at once (i.e. a single verification run from the entry point to all reachable states)
- We need: decomposition and verification of smaller units

Idea

1. Decompose the program into functions
2. Analyse each functions in isolation
3. Remember an overapproximation of the function's behavior, use it in further analysis instead of the function itself
   - function summaries [1]

[1] Patrice Godefroid, Compositional Dynamic Test Generation.

Idea

1. Decompose the program into functions
2. Analyse each functions in isolation
3. Remember an overapproximation of the function's behavior, use it in further analysis instead of the function itself
   - function summaries [1]

- Warning: most of this work exists mainly in minds of Mornfall and me. Rough edges and outright problems lie ahead.

[1] Patrice Godefroid, Compositional Dynamic Test Generation.

Outline

1. Compute the call graph of module M under analysis
2. Let M' be a topologically sorted SCC decomposition of M^R
   - a node can now have more than one entry point
3. Compute a summary s for each unit entry point f in M'
   - further analysis uses s on call-sites where f would be called

Call graph decomposition

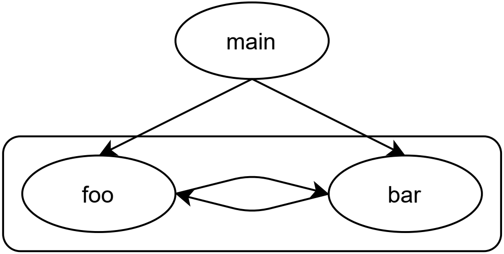
Example

```
int main() {
  return foo(5) + bar(4);
}

int foo(int i) {
  if (i == 0)
    return 0;
  return 1 + bar(i - 1);
}

int bar(int i) {
  if (i == 0)
    return 0;
  return 1 + foo(i - 1);
}
```

Call graph decomposition



Remark: Craig interpolation

Definition

Let (A, B) be a pair of formulae such that $A \wedge B$ is unsatisfiable. We say that formula I is an (A, B)–*interpolant* if the following properties hold:

- $A \rightarrow I$,
- $I \wedge B$ is unsatisfiable,
- every variable in I occurs both in A and B. [2]

Theorem (Craig)

If (A, B) is a pair of first-order formulae such that $A \wedge B$ is unsatisfiable, an (A, B)–interpolant exists.

[2] K.L. McMillan, Applications of Craig Interpolants in Model Checking

Summary computation

- Let $f : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$ be a function
- We want to summarise f, we will use interpolation

[3] Technically $x_1 \vee \neg x_1 \vee x_2 \vee x_3 \vee \ldots \vee x_n$

Summary computation

- Let $f : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$ be a function
- We want to summarise f, we will use interpolation

1. Assume a trivial precondition $A \leftarrow \top$ [3]
2. Set all parameters to unbounded symbolic terms (x_1, x_2, ...)
3. Run DIVINE on $f(x_1, x_2, \ldots)$. If no error $\rightarrow$ done
4. If failed, extract the path condition B from the error state
5. Set $A \leftarrow \mathit{interpolate}(\neg B, A)$
6. Repeat 3 with the new precondition

[3] Technically $x_1 \vee \neg x_1 \vee x_2 \vee x_3 \vee \ldots \vee x_n$

Summary computation

- If we get from A to B, $A \rightarrow B$ holds, $A \wedge \neg B$ is unsatisfiable
  - we can compute the interpolant
- The new precondition is built so that it disallows the counterexample (and does not allow previous counterexamples)
  - eventually converges to a precondition that disallows all error runs
- Once a correct precondition is computed, a constraint on the return value can be determined
  - this gives us the summary

Interesting parameters

- Not every function contains a reachable error
- Not every parameter can influence error manifestation
  - need to determine which parameters are interesting

Interesting parameters

- Not every function contains a reachable error
- Not every parameter can influence error manifestation
  - need to determine which parameters are interesting

Definition

Let f be a function with parameter x. Parameter x is interesting if there are values x_1, x_2 such that:

- $f(x_1)$ returns successfully,
- $f(x_2)$ yields an error.

Interesting parameters

Example

```
void foo(int i) {
    assert(i == 1);
}
```

i is an interesting parameter

Example

```
void foo(int i) {
    assert(false);
}
```

no interesting parameters

Example

```
int foo(int i, int j) {
    assert(i > 0);
    return i * j;
}
```

only *i* is an interesting parameter

Example

```
int foo(int i, int j) {
    int res = i * j;
    assert(res > 0);
    return res;
}
```

both parameters are interesting

Interesting sets of parameters

- We compute function preconditions
- Uninteresting parameters can be ignored
- We have to find the greatest set of interesting parameters
  - Systematically explore all partitions of parameters into interesting ($+$) and uninteresting ($-$)
  - Track which parameters participate in which values
  - $-$ parameters stop exploration on branches
  - If an error is found when flipping from $-$ to $+$, a parameter is interesting (not necessarily the one flipped)

$$\begin{tikzcd}[column sep=small]
 & +,+ \arrow[dl, no head] \arrow[dr, no head] & \\
+,- \arrow[dr, no head] & & -,+ \arrow[dl, no head] \\
 & -,- &
\end{tikzcd}$$

Implementation state

- We can use the LA LA Land infrastructure to find interesting parameters
  - Abstract domains *unit*, *counit*, *idtrack*
  - This part is done
- Everything happens on LLVM bitcode layer
  - a tool named SHOOP[4]

[4] No one remembers what the abbreviation means

Problems and future directions

- Good interpolants in bitvector logic — an open problem [5]
- Interaction of decomposition and parallel programs — possibly problematic
- Loss of precision in summarization — maybe some kind of refinement is needed [6]
- So far only functions with *n* inputs and a single output
  - Need to analyse pointer arguments and determine *in*/*out* usage
  - Global state is another problem
- Possible use for test synthesis

[5] Alberto Griggio, Effective Word-Level Interpolation for Software Verification

[6] Ondrej Sery et al., Interpolation-Based Function Summaries in Bounded Model Checking

Summary

- Modular approach to verification in DIVINE using function summaries through interpolation
- Still in early implementation phase
- Conceptual problems need to be overcome
  - mainly precise usage of interpolants and effective interpolant generation in bitvector logic
- In the future, hopefully another helpful part of DIVINE