Using LLVM in a Model Checking Workflow

Gyula Sallai

2018 European LLVM Developers Meeting
Introduction
Motivation

- Embedded software systems
  - Usually written in C
- Confidence in correctness?

\[ \neg (\text{Red} \land \text{Green}) \]
Software model checking

- Automatic transformation from source code

- Model checking is computationally hard
  - Undecidable in general
  - Model size/complexity must be reduced
LLVM for model checking

- LLVM IR as a language frontend?
  - Language-agnostic
  - Optimization infrastructure

- Using LLVM IR for model checking

\[\text{C} \rightarrow \text{LLVM IR} \rightarrow \text{Formal model} \rightarrow \text{theta framework}\]

\[\text{optimizations}\]

\[\text{C++} \rightarrow \text{LLVM IR} \rightarrow \text{Formal model} \rightarrow \text{Verification backend}\]

1https://github.com/ftsrg/theta
Transformation to formal models
Formal model for computer programs

- Control flow automata (CFA)

```c
int i = 0;
int sum = 0;
while (i < 11) {
    sum = sum + i;
    i = i + 1;
}
assert(i == 11);
```

- **error**: failing assertions
LLVM IR to formal models

- Gap between the IR and formal models
  - Designed for compilation ⇔ designed for theorem provers

- LLVM IR has more expressive power
  - SSA, ϕ-nodes → transformation rules
  - Pointers → theory of arrays, integer addresses
  - Global variables → promotion to locals
  - Procedure calls → function inlining
LLVM IR to formal models

bb0:
\[ x_0 = \text{call read()} \]
\[ \text{br}(\text{incr, bb1, bb2}) \]

bb1:
\[ x_1 = x_0 + 1 \]

bb2:
\[ x_2 = x_0 - 1 \]

bb3:
\[ x_4 = \phi(\{x_1, bb1\}, \{x_2, bb2\}) \]

...
Optimization algorithms
Optimizations

- Need to be configurable

- Optimizations in LLVM
  - Constant propagation, dead code elimination
  - Function inlining

- Other transformations
  - Global variables to locals
  - Program slicing
Program slicing

- **Slice**: subprogram, which produces the same output and assigns the same values to a set of variables as the original program.

```c
0: int i = 0;
1: int x = 0;
2: while (i < 11) {
3:     x = x + i;
4:     i = i + 1;
}
5: assert(i != 0);
```

 Criterion: value of $i$ at statement 5

```c
0: int i = 0;
1: int x = 0;
2: while (i < 11) {
3:     x = x + i;
4:     i = i + 1;
}
5: assert(i != 0);
```
Evaluation
Evaluation

- **SV-Comp: Competition on Software Verification**\(^1\)
  - Verification tasks written in C

- **Program categories**
  - *locks*: locking mechanisms
  - *eca*: event-driven systems
  - *ssh*: ssh protocol

---

\(^1\) [https://sv-comp.sosy-lab.org/2016/](https://sv-comp.sosy-lab.org/2016/)
Evaluation

*Opt: with optimizations
*Slice: with slicing

<table>
<thead>
<tr>
<th>Model</th>
<th>Vars</th>
<th>Locs</th>
<th>VarsOpt</th>
<th>LocsOpt</th>
<th>#Slice</th>
<th>VarSlice</th>
<th>LocsSlice</th>
</tr>
</thead>
<tbody>
<tr>
<td>locks10</td>
<td>55</td>
<td>236</td>
<td>52</td>
<td>231</td>
<td>10</td>
<td>5.5</td>
<td>27</td>
</tr>
<tr>
<td>locks14</td>
<td>75</td>
<td>324</td>
<td>72</td>
<td>319</td>
<td>14</td>
<td>5.5</td>
<td>26.5</td>
</tr>
<tr>
<td>eca1</td>
<td>1104</td>
<td>2937</td>
<td>976</td>
<td>2870</td>
<td>1</td>
<td>614</td>
<td>1908</td>
</tr>
<tr>
<td>eca2</td>
<td>1040</td>
<td>2854</td>
<td>892</td>
<td>2778</td>
<td>1</td>
<td>590</td>
<td>1936</td>
</tr>
<tr>
<td>eca3</td>
<td>3269</td>
<td>10719</td>
<td>2781</td>
<td>10325</td>
<td>1</td>
<td>2408</td>
<td>9050</td>
</tr>
<tr>
<td>ssh1</td>
<td>196</td>
<td>693</td>
<td>174</td>
<td>648</td>
<td>1</td>
<td>109</td>
<td>394</td>
</tr>
</tbody>
</table>

Many small slices

Some reduction with optimizations, more with slicing

Significant reduction
Summary

- Software model checking
- LLVM IR-based model checking
  - Transformation to formal models
  - Configurable optimizations
  - Program slicing
- Future work
  - Improved pointer support
  - New slicing methods (heuristics...)

CFA error is not reachable

Model checking

C code

OK

Counterexample

0: int i = 0;
2: while (i < 11)
   4:   i = i + 1;
5: assert(i != 0);