Modular Bounded Model Checking for Software Verification

Marko Kleine Büning

Institute for Theoretical Computer Science
Reliable Software Systems in the Automotive Industry

COOP 2022
Motivation: Software Verification of Industrial Software

- Embedded software is for example deployed in automotive, aerospace and medical devices
- In such safety-critical areas software has to meet the highest quality standards
- Errors can have catastrophic consequences:

Arianne 5 Flight 501
June 1996

Toyota Camry 2005
September 2007
Motivation: Software Verification of Industrial Software

- Embedded software is for example deployed in automotive, aerospace and medical devices
- In such safety-critical areas software has to meet the highest quality standards
- Errors can have catastrophic consequences:

  Uber crash with pedestrian
  March 2018

  Tesla Model S3 Crash
  Mai 2018
State-of-the-Art: Bounded Model Checking (BMC)

- Analyze all inputs of the program and all paths up to a fixed length $b$
- Software Bounded Model Checking as implemented in tool LLBMC [1]
The Scalability Problem of Bounded Model Checking

- For BMC a program under verification is encoded into a logical formula.
- Even when ignoring time constraints, the memory requirements to encode millions of lines of code is not attainable.

Modern cars: **100 MLoc**

Current audio control: **4 MLoc**

Cars in the next 3 years: **300 MLoc**
A Possible Solution: Modularization

- Partition program into modules that can then be solved individually
- Modularization requires formalization of interfaces and dependencies
- There exist several approaches:
  - Compositional Verification [2]
  - Assume-guarantee reasoning [3]

- Number of lines of specification for one line of source code:
  - 2 for specialized [4]
  - 5 for SMT-based [5]
  - up to 20 for interactive theorem prover approaches [6]

→ The aim is to create a fully automatic modularization for bounded model checking of industrial sized software
Our Solution: Modular Software Verification

[Diagram of modular software verification process]

Kleine Büning: Modular Bounded Model Checking for Software Verification
Set-Up

Program $P$

Compilation → Check-Insertion → Configuration → Preprocessing → (Global Analysis)

Set-Up

Program $P'$ → Checks

[KB5]
Modularization

- Program $P'$
- Checks
- Modularization
- Function Call Abstraction
- Environment Abstraction
- Modules $\{M_c\}$

[KB4]
Decomposition of Programs - Idea

- Modularization approach that does not require manual specifications
- Over-approximation (i.e. abstraction) of the original program
  - set of program traces that we check for a module are superset of traces present in original program
- Often over-approximations of variable values - such as in abstract interpretation [7]
- Our approach: abstraction of **structural kind**
Call Graph of Program under Verification

- Program consisting of four functions and four checks to be verified:
- Nodes: functions
- Edges: function calls
- Green arrow: entry-point function
- Green triangle: check
- Green border: module under verification
Decomposition – Function Call Abstraction

- Fully over-approximates calls to functions outside of the chosen module

\[ \tau_{y=\text{call}} f^*(x_1, \ldots, x_n)(v, m, l) = \]
\[ \{(v[y^l \leftarrow i], m', \text{next}(l)) \mid i \in Val, m' \in (\text{Addr} \rightarrow Val)\} \]
Decomposition – Function Call Abstraction

- Fully over-approximates calls to functions outside of the chosen module
Decomposition – Environment Abstraction

- Fully over-approximates the call environment of the verified module

\[ I' = \{ (v, m, l) | \]

\[ v(g) = \text{address of global variable } g \text{ for all globals} \]

\[ v(x^l) = \text{arbitrary value for local variable } x \text{ for all local vars.} \]

\[ v(p^l_k) = \text{arbitrary value for parameter } k \text{ of function } f \]

\[ m : \text{any function } \text{Adr} \rightarrow \text{Val} \]

\[ l = (f, bb_{\text{Entry}}, 0) \]

\}
Decomposition – Environment Abstraction

- Fully over-approximates the call environment of the verified module
Combination and Result of Abstractions

**Advantage:**
- We can generate modules containing only a single function and can generate modules of arbitrary size by combining both abstractions.

**Disadvantage:**
- Over-approximation of call environment and function calls potentially leads to *false positives* – errors messages where there are no errors.

**Open Challenge:**
- Balancing of precision and scalability through refined abstractions.
Refinement

Kleine Büning: Modular Bounded Model Checking for Software Verification
Example – Whole Program Verification

```c
void top(int x, int y) {
    if (x > 0) {
        x = x - 1;
        mid1(x, y);
        mid2(x, y);
    }
}

void mid1(int x, int y) {
    if (x > 0) {
        x = x - 1;
        bot(x, y);
    }
}

void mid2(int x, int y) {
    if (y > 100) {
        bot(x, y);
    }
}

void bot(int x, int y) {
    if (y >= 0 && y < 100) {
        // assert(x < 2147483646);
        x = x + 2;
    }
}
```
Decomposition – Call Environment Abstraction

```java
void top(int x, int y) {
    if (x > 0) {
        x = x - 1;
        mid1(x, y);
        mid2(x, y);
    }
}

void mid1(int x, int y) {
    if (x > 0) {
        x = x - 1;
        bot(x, y);
    }
}

void mid2(int x, int y) {
    if (y > 100) {
        bot(x, y);
    }
}

void bot(int x, int y) {
    if (y >= 0 && y < 100) {
        // assert(x < 2147483646);
        x = x + 2;
    }
}
```
**Refinement – Preconditions Substitution**

- Generate a precondition representing the erroneous input space

- For function `bot(int x, int y)` such a precondition would be:
  - $x < 2147493646 \lor y < 0 \lor y \geq 100$

- Calls of function `bot` can then be substituted with this precondition

```java
void bot(int x, int y){
    if( y >= 0 && y < 100 ) {
        // assert(x < 2147483646);
        x = x + 2 ;
    }
}
```
Refinement – Preconditions Substitution

```c
void top(int x, int y) {
    if (x > 0) {
        x = x - 1;
        mid1(x, y);
        mid2(x, y);
    }
}

void mid1(int x, int y) {
    if (x > 0) {
        x = x - 1;
        @assert.precondition_bot(x, y);
    }
}

void mid2(int x, int y) {
    if (y > 100) {
        @assert.precondition_bot(x, y);
    }
}

void bot(int x, int y) {
    if (y >= 0 && y < 100) {
        // assert(x < 2147483646);
        x = x + 2;
    }
}
```
Refinement – Preconditions Substitution

```c
void top(int x, int y){
    if(x > 0) {
        x = x - 1;
        @assert.precondition_mid1(x, y);
        @assert.safe();
    }
}

void mid1(int x, int y){
    if(x > 0){
        x = x - 1;
        @assert.precondition_bot(x, y);
    }
}

void mid2(int x, int y){
    if(y > 100){
        @assert.precondition_bot(x, y);
    }
}

void bot(int x, int y){
    if( y >= 0 && y < 100) {
        // assert(x < 2147483646);
        x = x + 2 ;
    }
}
```
Refinement – Precondition Generation

Enumerative Preconditions:
- Based on the counterexamples generated by the bounded model checker
- In our example: 200 (2 values for x times 100 values for y) data points
- Combination represents the enumerative precondition
- Part of data points represent under-approximated precondition

Learned Preconditions:
- Generalize the under-approximated precondition
- Generate negative and positive data points and employ an IC3-based learner

```c
void bot(int x, int y){
    if( y >= 0 && y < 100 ) {
        // assert(x < 2147483646);
        x = x + 2 ;
    }
}
```
Evaluation Results – BMI160 Driver

- Open-source library BMI160-Driver is a low power control unit driver
- Implemented by Robert Bosch GmbH with around 2000 LoC plus libraries

<table>
<thead>
<tr>
<th>BMI160-Driver</th>
<th>safe</th>
<th>loop-bound safe</th>
<th>cond. unsafe</th>
<th>unsafe</th>
<th>unknown</th>
<th>Time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>not applicable due to missing main function</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I-S-Mod-EA</td>
<td>445 (78%)</td>
<td>9(2%)</td>
<td>120 (21%)</td>
<td>0(0%)</td>
<td>0 (0%)</td>
<td>112</td>
</tr>
<tr>
<td>C-Mod-EA-FC</td>
<td>431 (75%)</td>
<td>22(4%)</td>
<td>100 (17%)</td>
<td>0(0%)</td>
<td>21(4%)</td>
<td>268</td>
</tr>
<tr>
<td>C-Mod-EA</td>
<td>349 (61%)</td>
<td>141(25%)</td>
<td>84 (15%)</td>
<td>0(0%)</td>
<td>0 (0%)</td>
<td>360</td>
</tr>
<tr>
<td>I-Mod-EA-FC</td>
<td>441 (77%)</td>
<td>22(4%)</td>
<td>109 (19%)</td>
<td>0(0%)</td>
<td>2 (0%)</td>
<td>702</td>
</tr>
<tr>
<td>I-Mod-EA</td>
<td>349 (61%)</td>
<td>140(24%)</td>
<td>84 (15%)</td>
<td>0(0%)</td>
<td>1 (0%)</td>
<td>798</td>
</tr>
<tr>
<td>I-Mod-EA-Ref-EP(5)</td>
<td>386 (67%)</td>
<td>134(23%)</td>
<td>49 (9%)</td>
<td>5(1%)</td>
<td>0 (0%)</td>
<td>899</td>
</tr>
<tr>
<td>I-Mod-EA-Ref-CA(1)</td>
<td>378 (66%)</td>
<td>136(24%)</td>
<td>55 (10%)</td>
<td>5(1%)</td>
<td>0 (0%)</td>
<td>8791</td>
</tr>
<tr>
<td>I-Mod-EA-Ref-CA(5)</td>
<td>497 (87%)</td>
<td>32(6%)</td>
<td>5 (1%)</td>
<td>40(7%)</td>
<td>0 (0%)</td>
<td>16485</td>
</tr>
<tr>
<td>I-Mod-EA-Ref-C(5)-EP(5)</td>
<td>501 (87%)</td>
<td>20(3%)</td>
<td>13 (2%)</td>
<td>40(7%)</td>
<td>0 (0%)</td>
<td>24114</td>
</tr>
</tbody>
</table>
Evaluation Results – BMI160 Driver

- Comparison with other tools is difficult, because of checked properties

- Results of verifying BMI160 with the Abstract Interpretation tool Polyspace 2017b:

<table>
<thead>
<tr>
<th>Polyspace</th>
<th>green</th>
<th>orange</th>
<th>red</th>
<th>grey</th>
<th>Total</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMI160</td>
<td>45 (19%)</td>
<td>142 (61%)</td>
<td>0 (0%)</td>
<td>46 (20%)</td>
<td>233</td>
<td>988</td>
</tr>
</tbody>
</table>

- Results of verifying BMI160 with our Enumerative Precondition approach:

<table>
<thead>
<tr>
<th>Ref-EP(5)</th>
<th>safe</th>
<th>loop-bound safe</th>
<th>cond. unsafe</th>
<th>unsafe</th>
<th>unknown</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMI160</td>
<td>386 (67%)</td>
<td>134 (23%)</td>
<td>49 (9%)</td>
<td>5 (1%)</td>
<td>0 (0%)</td>
<td>899</td>
</tr>
</tbody>
</table>
Evaluation Results – BMI160 Driver

- Comparison with other tools is difficult, because of checked properties.

- Results of verifying BMI160 with the Abstract Interpretation tool Polyspace 2017b:

<table>
<thead>
<tr>
<th>Polyspace</th>
<th>green</th>
<th>orange</th>
<th>red</th>
<th>grey</th>
<th>Total</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMI160</td>
<td>45 (19%)</td>
<td>142 (61%)</td>
<td>0 (0%)</td>
<td>46 (20%)</td>
<td>233</td>
<td>988</td>
</tr>
</tbody>
</table>

- Results of verifying BMI160 with our Enumerative Precondition approach:

<table>
<thead>
<tr>
<th>Ref-EP(5)</th>
<th>safe</th>
<th>loop-bound safe</th>
<th>cond. unsafe</th>
<th>unsafe</th>
<th>unknown</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMI160</td>
<td>386 (67%)</td>
<td>134 (23%)</td>
<td>49 (9%)</td>
<td>5 (1%)</td>
<td>0 (0%)</td>
<td>899</td>
</tr>
</tbody>
</table>
## Evaluation Results – SQLite

- C-language library implementing SQL database engine
- Contains around 400K LOC plus libraries

<table>
<thead>
<tr>
<th>Approach/Results</th>
<th>safe</th>
<th>loop-bound safe</th>
<th>cond. unsafe</th>
<th>unsafe</th>
<th>unknown</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
<td>8371 (0%)</td>
<td>TO</td>
</tr>
<tr>
<td>I-S-Mod-EA</td>
<td>698 (8%)</td>
<td>2095 (25%)</td>
<td>3138 (37%)</td>
<td>0 (0%)</td>
<td>2426 (29%)</td>
<td>38'893</td>
</tr>
<tr>
<td>I-S-Mod-EA-FC</td>
<td>684 (8%)</td>
<td>2091 (25%)</td>
<td>3157 (37%)</td>
<td>0 (0%)</td>
<td>2439 (29%)</td>
<td>43'865</td>
</tr>
<tr>
<td>C-Mod-EA</td>
<td>42 (1%)</td>
<td>92 (1%)</td>
<td>42 (1%)</td>
<td>0 (0%)</td>
<td>8195 (98%)</td>
<td>529'222</td>
</tr>
<tr>
<td>I-S-Mod-EA-Ref-EP</td>
<td>1965 (23%)</td>
<td>791 (9%)</td>
<td>3135 (37%)</td>
<td>54 (1%)</td>
<td>2426 (29%)</td>
<td>445'147 (6d)</td>
</tr>
<tr>
<td>I-S-Mod-EA-Ref-CI</td>
<td>698 (8%)</td>
<td>2096 (25%)</td>
<td>2769 (33%)</td>
<td>369 (4%)</td>
<td>2439 (29%)</td>
<td>639'563 (8d)</td>
</tr>
</tbody>
</table>
Conclusion: Modular Software Verification

Kleine Büning: Modular Bounded Model Checking for Software Verification
Thank you! Any Questions?

Contact Information:
Marko Kleine Büning
Mail: marko.kleinebuing@kit.edu
Tel: +49721 608 45955
References


References


