Karlsruhe Institute of Technology

Modular Bounded Model Checking for Software Verification

Marko Kleine Büning

COOP 2022

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

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

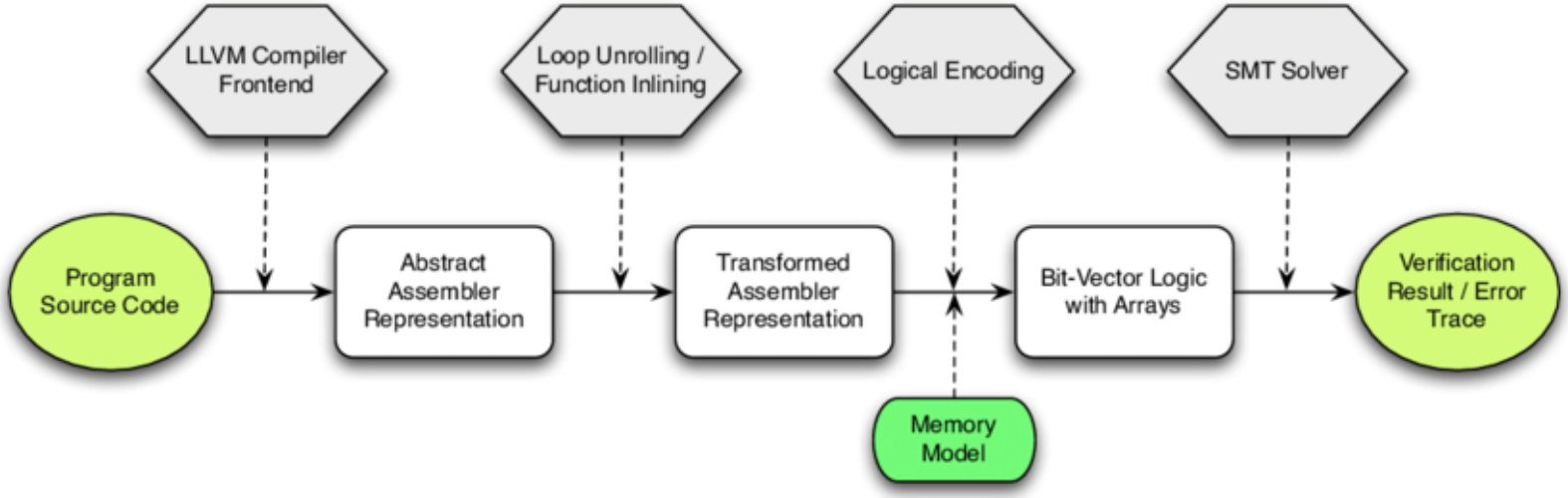
- Embedded software is for example deployed in automotive, aerospace and medical devices
- In such safety-critical areas software has to meet the highest quality standards
- Errors can have catastrophic consequences:

Uber crash with pedestrian
March 2018

Tesla Model S3 Crash
Mai 2018

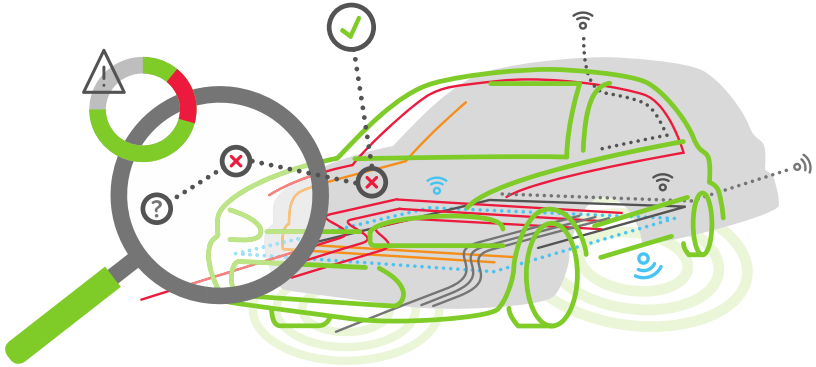
State-of-the-Art: Bounded Model Checking (BMC)

- 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 yeas: **300 MLoC**

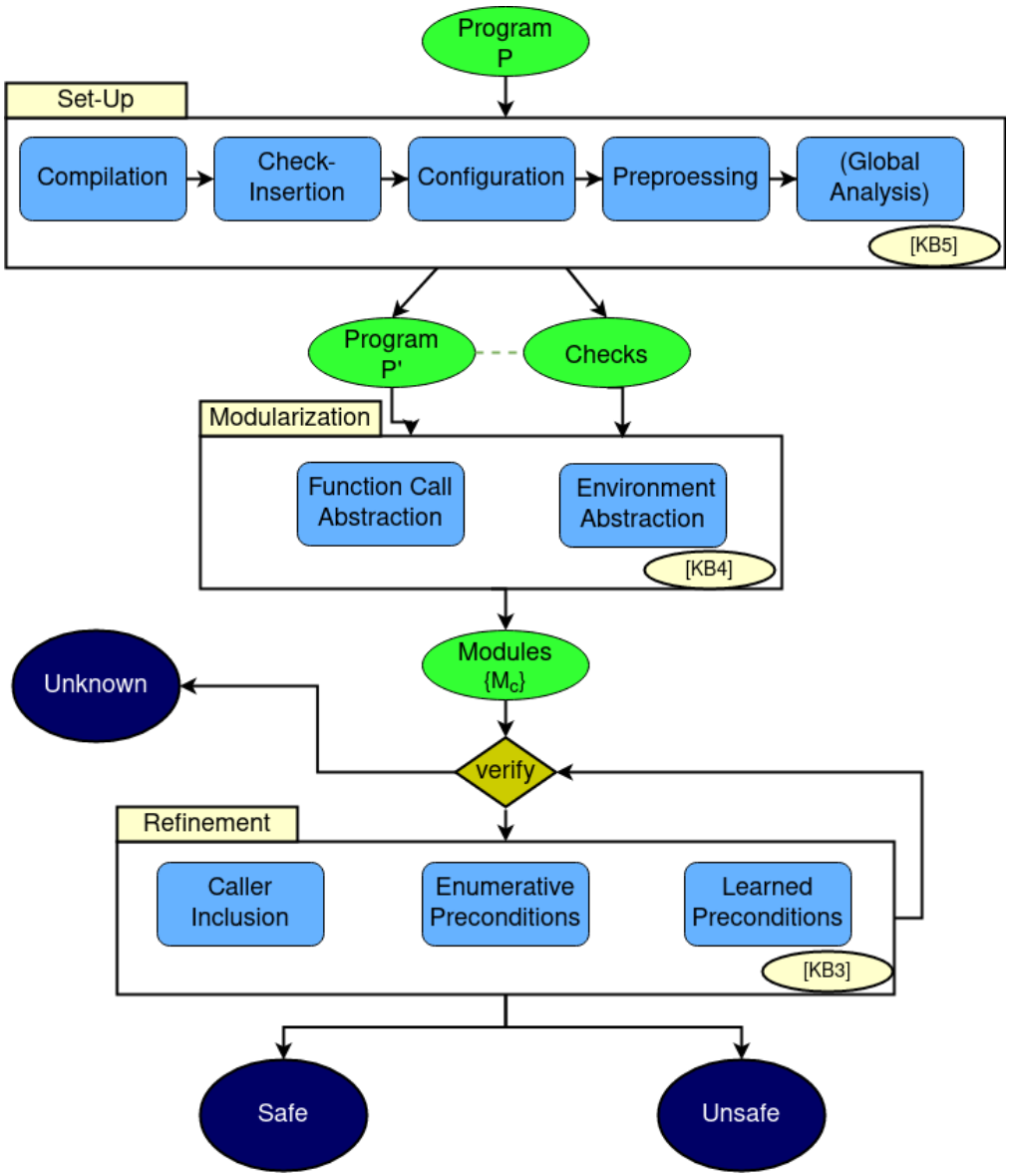
A Possible Solution: Modularization

- 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]

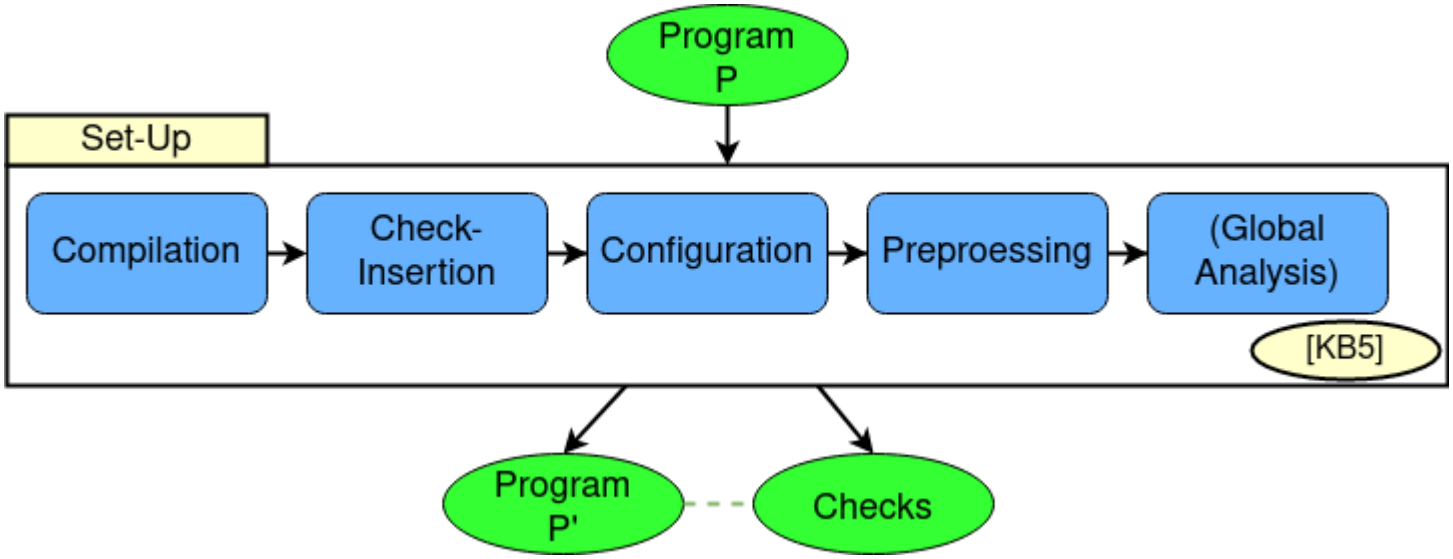
- Number of lines of specification for one line of source code:
  - 2 for specialized [4]
  - 5 for SMT-based [5]
  - up to 20 for interactive theorem prover approaches [6]

→ The aim is to create a fully automatic modularization for bounded model checking of industrial sized software

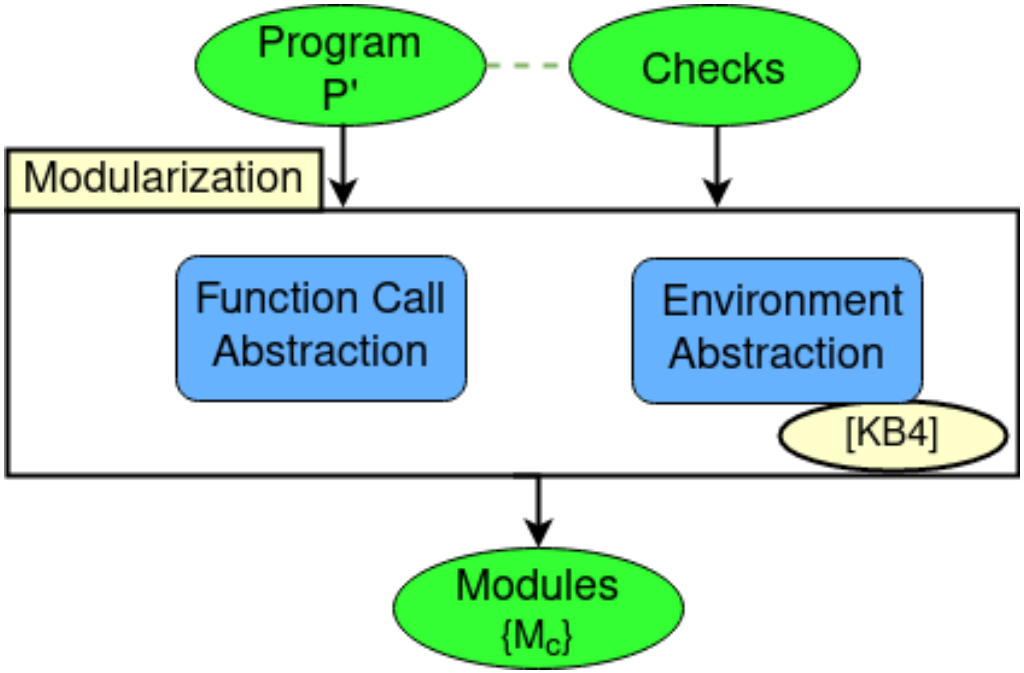
Our Solution: Modular Software Verification



Set-Up


Program
P
Set-Up
Compilation
Check-
Insertion
Configuration
Preproessing
(Global
Analysis)
[KB5]
Program
P'
Checks

Modularization



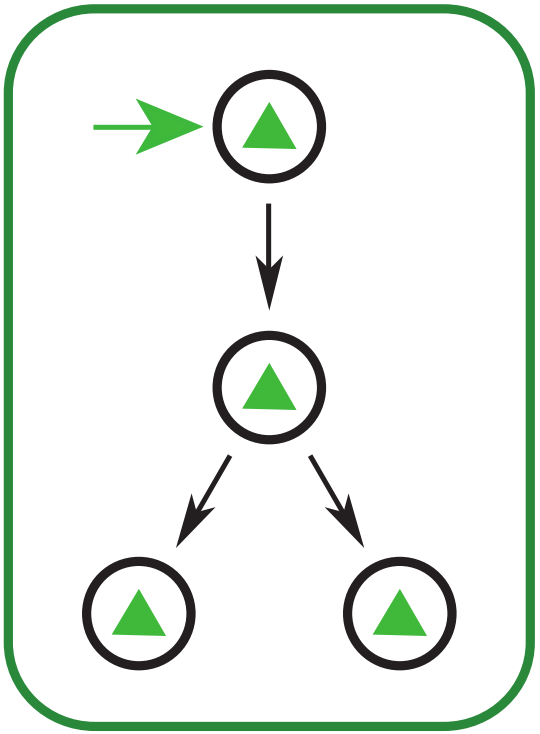
Decomposition of Programs - Idea

- 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:

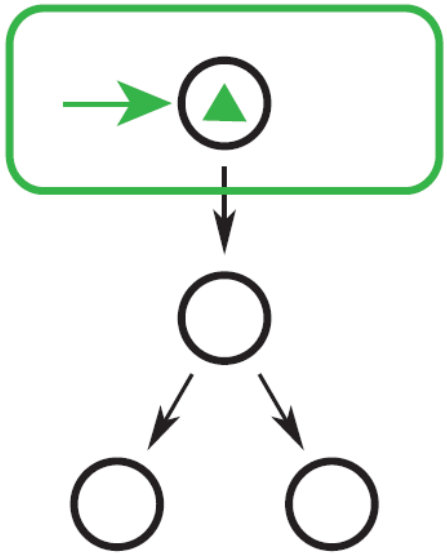
- Nodes: functions
- Edges: function calls
- Green arrow: entry-point function
- Green triangle: check
- Green border: module under verification



Decomposition – Function Call Abstraction

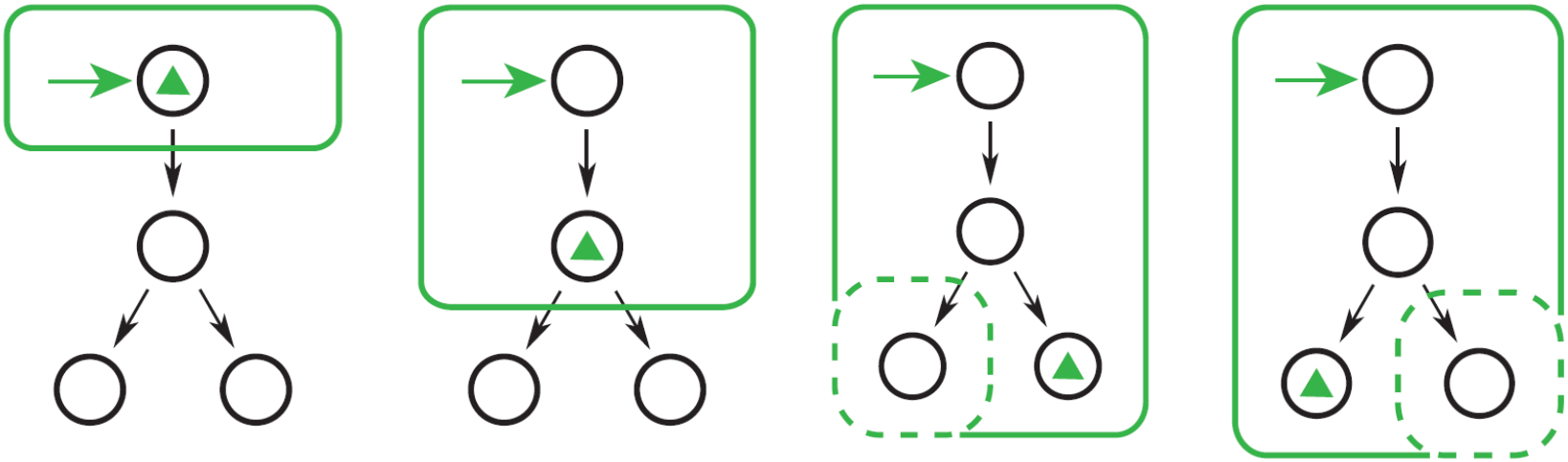
- Fully over-approximates calls to functions outside of the chosen module

$$\tau_{y=\texttt{call}\ f^*(x_1,\ldots,x_n)}(v,m,l) = \{(v[y^l \leftarrow i], m', next(l)) \mid i \in Val, m' \in (Adr \rightarrow Val)\}$$



Decomposition – Function Call Abstraction

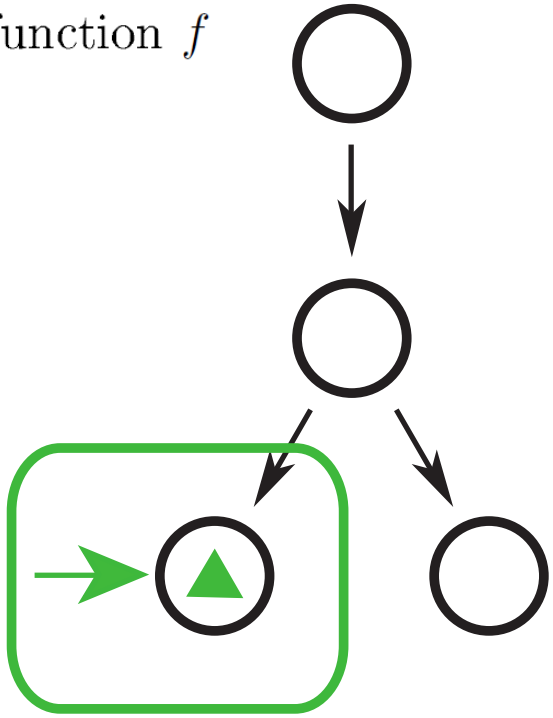
- Fully over-approximates calls to functions outside of the chosen module



Decomposition – Environment Abstraction

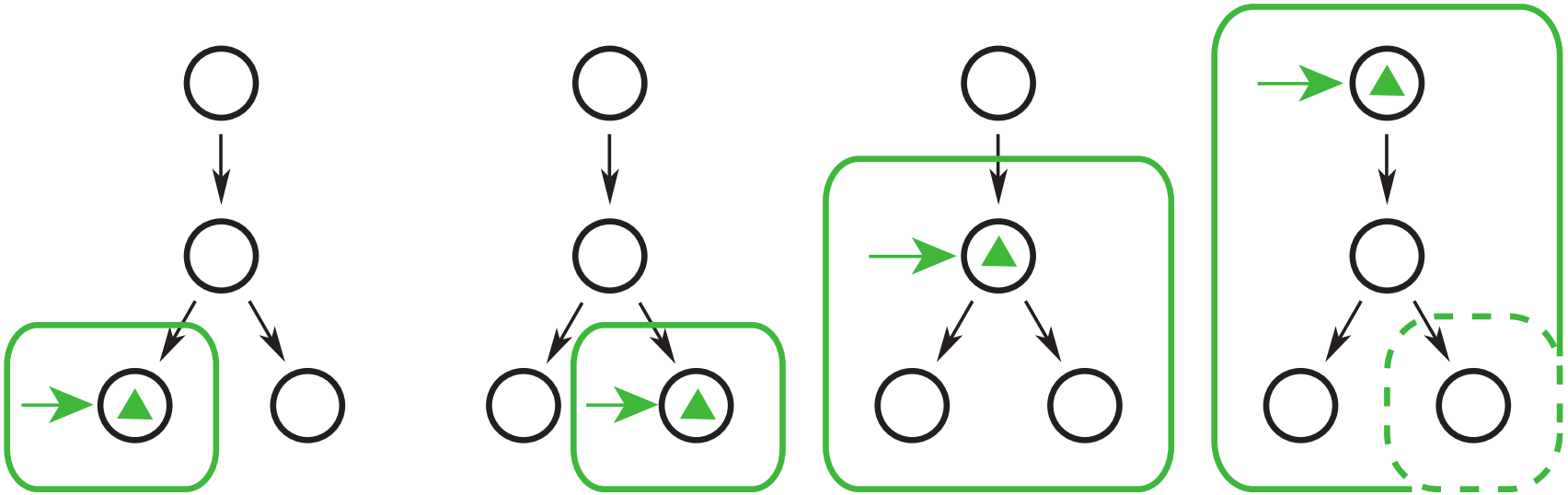
- Fully over-approximates the call environment of the verified module

$$
\begin{aligned}
I' = \ \{(v, m, l) \mid & \\
& v(g) = \text{address of global variable } g \text{ for all globals} \\
& v(x^l) = \text{arbitrary value for local variable } x \text{ for all local vars.} \\
& v(p_k^l) = \text{arbitrary value for parameter } k \text{ of function } f \\
& m : \text{any function } Adr \rightarrow Val \\
& l = (f, bb_{\texttt{Entry}}, 0) \quad \}
\end{aligned}
$$



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

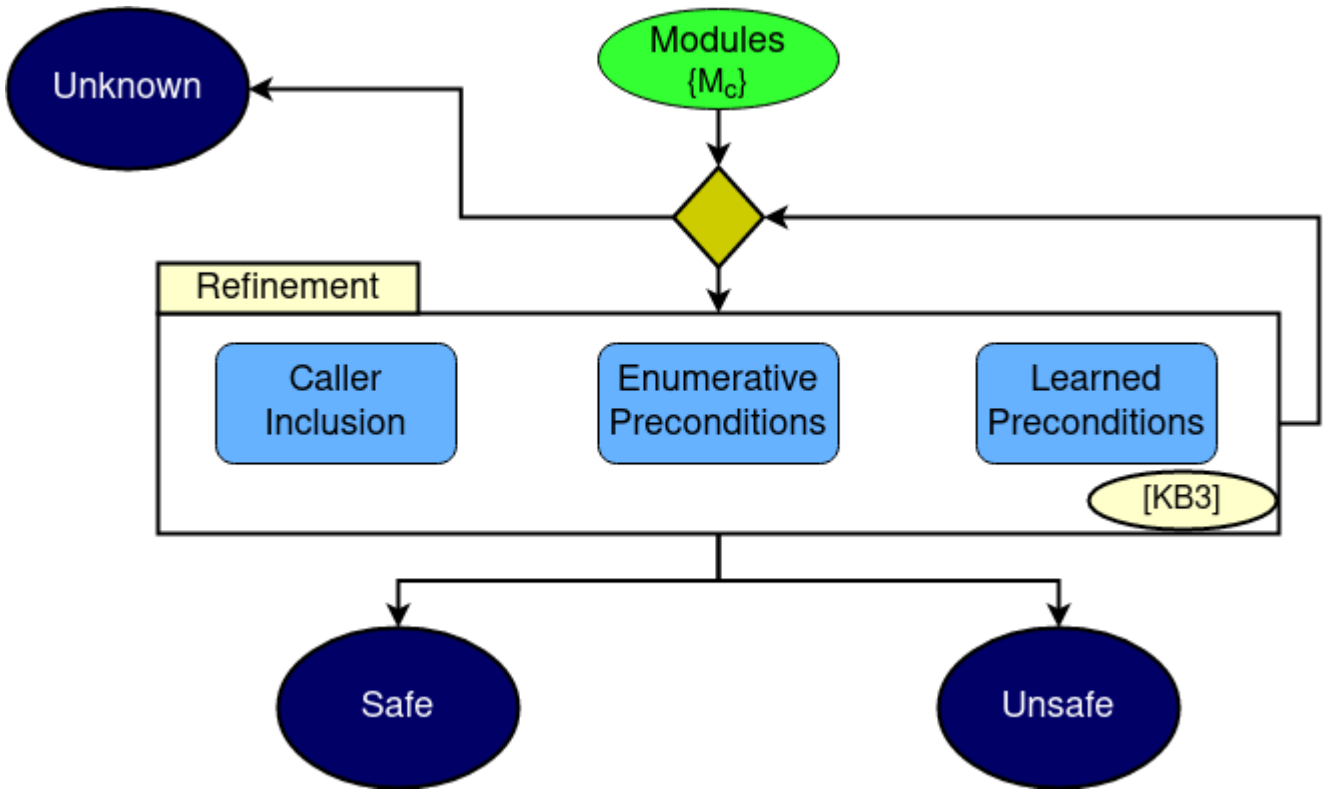
Disadvantage:

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

Open Challenge:

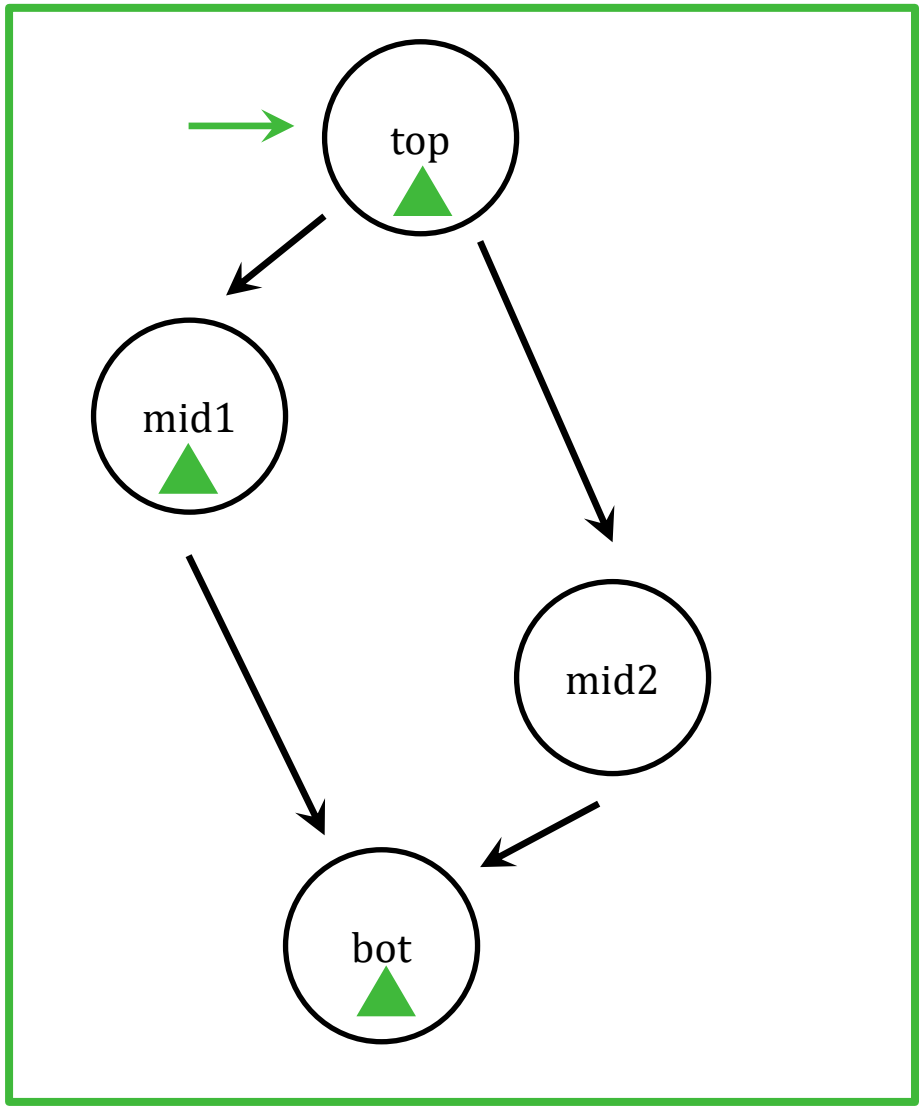
- Balancing of precision and scalability through refined abstractions

Refinement


Unknown
Modules
{Mc}
Refinement
Caller Inclusion
Enumerative Preconditions
Learned Preconditions
[KB3]
Safe
Unsafe

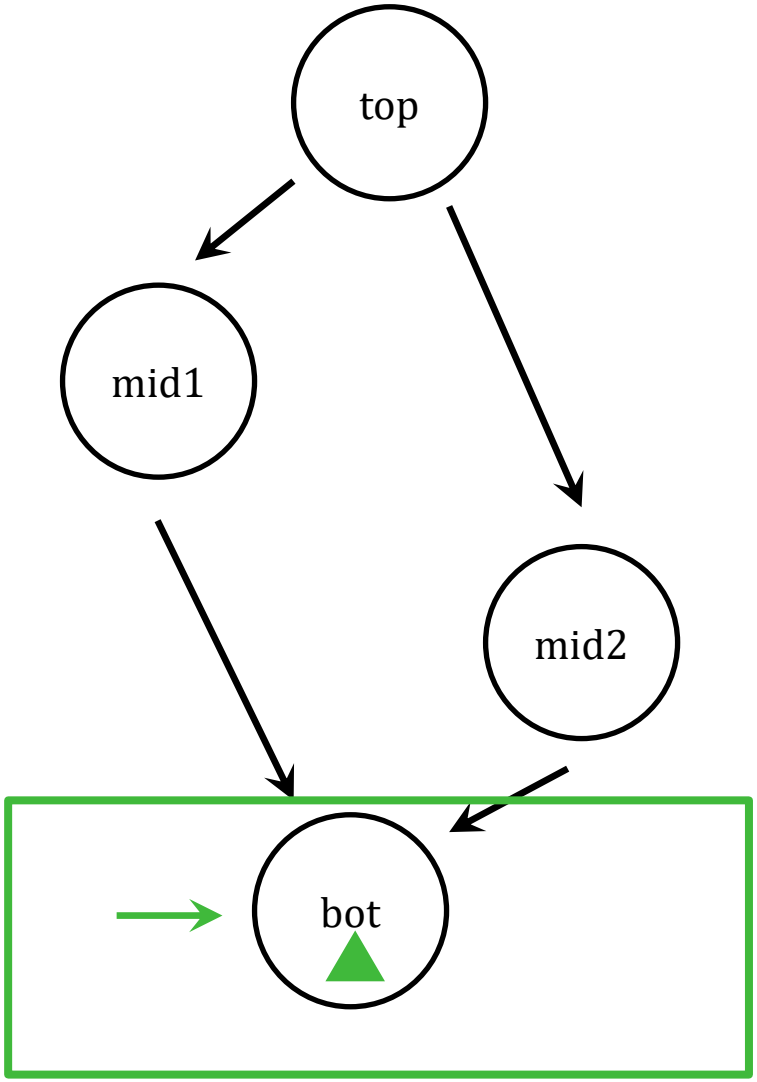
Example – Whole Program Verification

```
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

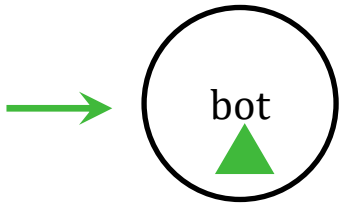
```
void top(int x , int y ){
    if(x > 0) {
        x = x - 1;
        mid1(x, y);
        mid2(x, y);
    }
}

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

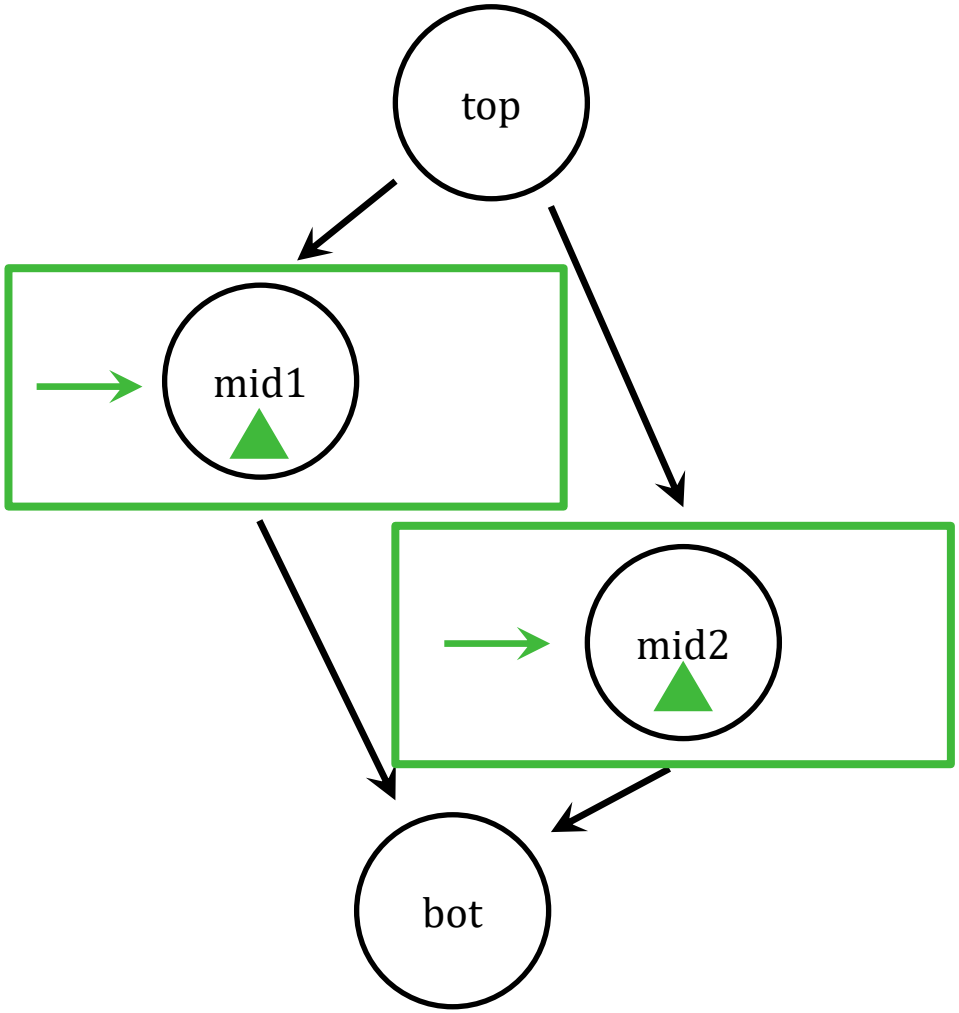
- Generate a precondition representing the erroneous input space
- For function **bot(int x, int y)** such a precondition would be:
  - $x < 2147493646 \vee y < 0 \vee y \geq 100$
- Calls of function **bot** can then be substituted with this precondition

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



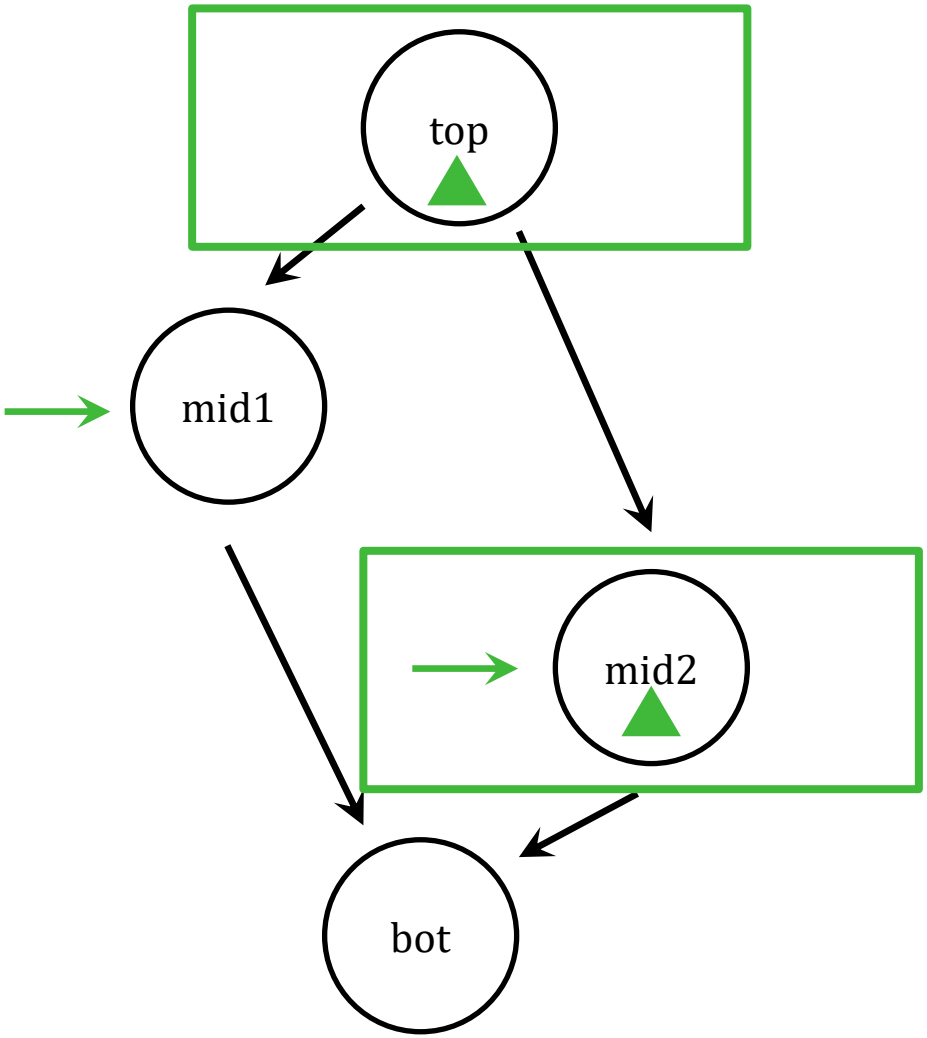
Refinement – Preconditions Substitution

```
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

```
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

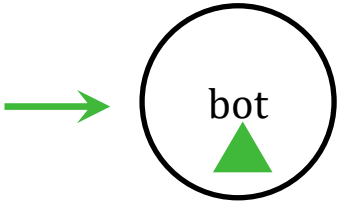
Enumerative Preconditions:

- 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

```
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

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

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**:

Polyspace	green	orange	red	grey	Total	Time
BMI160	45 (19%)	142 (61%)	0 (0%)	46 (20%)	233	988

- Results of verifying BMI160 with our Enumerative Precondition approach:

Ref-EP(5)	safe	loop-bound safe	cond. unsafe	unsafe	unknown	Time
BMI160	386 (67%)	134(23%)	49 (9%)	5(1%)	0 (0%)	899

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**:

Polyspace	green	orange	red	grey	Total	Time
BMI160	45 (19%)	142 (61%)	0 (0%)	46 (20%)	233	988

- Results of verifying BMI160 with our Enumerative Precondition approach:

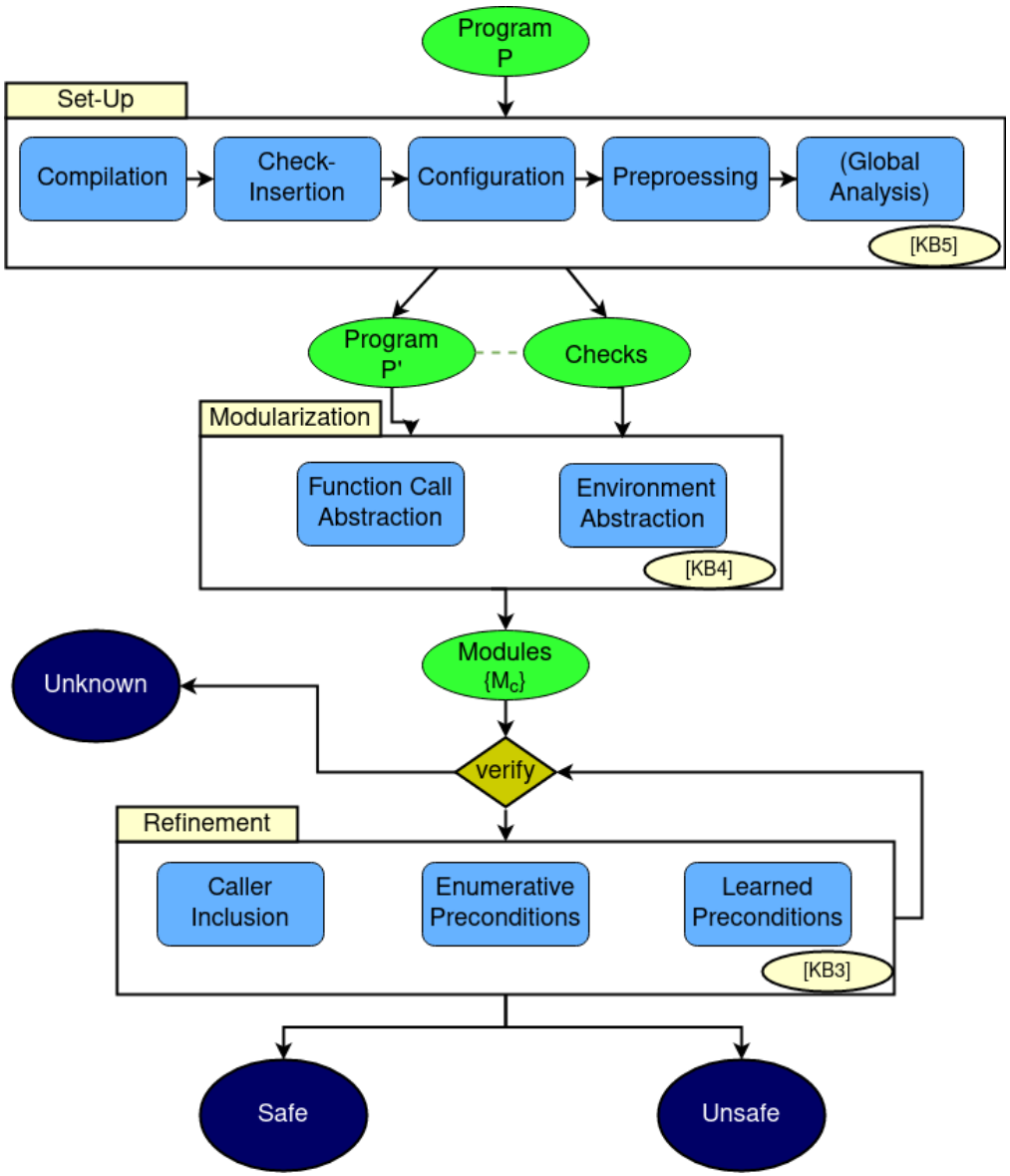
Ref-EP(5)	safe	loop-bound safe	cond. unsafe	unsafe	unknown	Time
BMI160	386 (67%)	134(23%)	49 (9%)	5(1%)	0 (0%)	899

Evaluation Results – SQLite

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

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

Conclusion: Modular Software Verification



Thank you! Any Questions?

Contact Information:

Marko Kleine Büning
Mail: marko.kleinebuening@kit.edu
Tel: +49721 608 45955

References

[1] Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Proceedings of the 4th International Conference on Veried Software: Theories, Tools, and Experiments (VSTTE '12). pp. 146161. Springer (2012)

[2] Clarke, E.M., Long, D.E., McMillan, K.L.: Compositional model checking. In: [1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science. pp. 353362. IEEE (1989)

[3] Grumberg, O., Long, D.E.: Model checking and modular verication. In: International Conference on Concurrency Theory. pp. 250265. Springer (1991)

[4] Müller, P.: The binomial heap verication challenge in Viper. In: Principled Software Development, pp. 203219. Springer (2018)

[5] Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill,B.: Ironclad apps: End-to-end security via automated full-system verication. In:11th Symposium on Operating Systems Design and Implementation. pp. 165181(2014)

[6] Kaiser, J.O., Dang, H.H., Dreyer, D., Lahav, O., Vafeiadis, V.: Strong logic for weak memory: Reasoning about release-acquire consistency in Iris. In: 31st European Conference on Object-Oriented Programming (ECOOP 2017). Schloss Dagstuhl- Leibniz-Zentrum fuer Informatik (2017)

[7] Cousot, Patrick. "Abstract interpretation." ACM Computing Surveys (CSUR) 28.2 (1996): 324-328.

References

[KB1] Marko Kleine Büning, Tomáš Balyo, and Carsten Sinz. “Using DimSpec for Bounded and Unbounded Software Model Checking”. In: International Conference on Formal Engineering Methods (ICFEM). Springer. 2019, pp. 19–35.

[KB2] Marko Kleine Büning, Philipp Kern, and Carsten Sinz. “Verifying equivalence properties of neural networks with relu activation functions”. In: International Conference on Principles and Practice of Constraint Programming (CP). Springer. 2020, pp. 868–884.

[KB3] Marko Kleine Büning, Johannes Meuer, and Carsten Sinz. Refined Modularization for Bounded Model Checking by Precondition Generation. Manusscript in preparation for submission. 2022.

[KB4] Marko Kleine Büning and Carsten Sinz. “Automatic modularization of large programs for bounded model checking”. In: International Conference on Formal Engineering Methods (ICFEM). Springer. 2019, pp. 186–202.

[KB5] Marko Kleine Büning, Carsten Sinz, and David Faragó. “QPR Verify: A Static Analysis Tool for Embedded Software Based on Bounded Model Checking”. In: Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE). Springer, 2020, pp. 21–32..