2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION

UNITED STATES

SAN JOSE, CA, USA
MARCH 4-7, 2024

Automating the Formal Verification of Firmware: A Novel Foundation and Scalable Methodology

Bryan Olmos, Sanjana Sainath, Wolfgang Kunz, Djones Lettnin

Infineon

RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

accellera SYSTEMS INITIATIVE

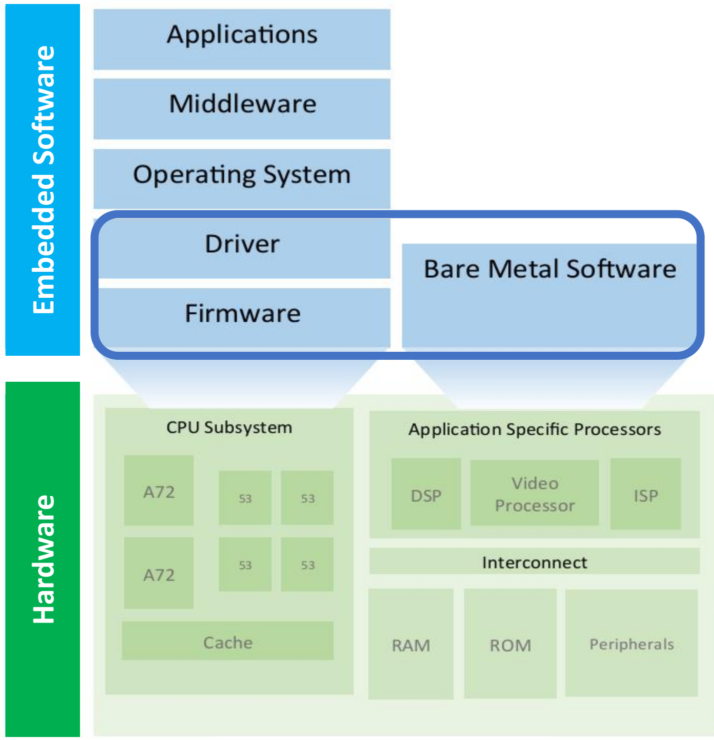
Motivation of the Paper

- Increase the reliability of firmware designs based on C code
- Help meet industry standards
- Reducing costs by catching problems earlier

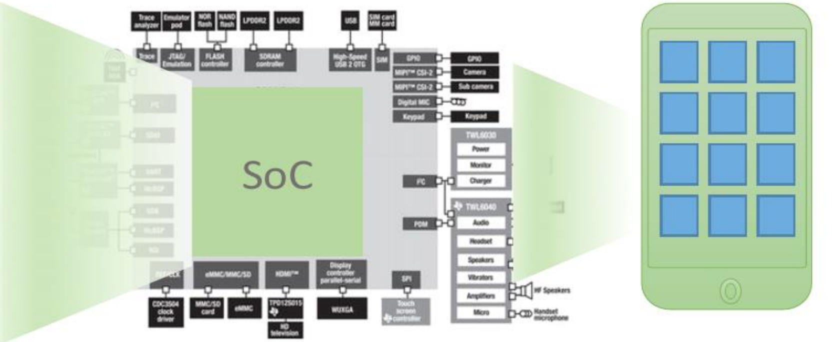
Software bugs	Consequences
Tesla recalls almost 12k vehicles, 2021	A glitch in its Full-Self Driving software
T-Mobile data breach, 2021	Affects 50 million customers
Amazon AWS Outage, 2017	Problems for hundreds of websites

Scope – Target Software

- This paper verifies software used to control hardware devices



- Property Verification
- Weaknesses Detection
- Code Coverage



Integrated Circuit Level

Board Level

Product Level

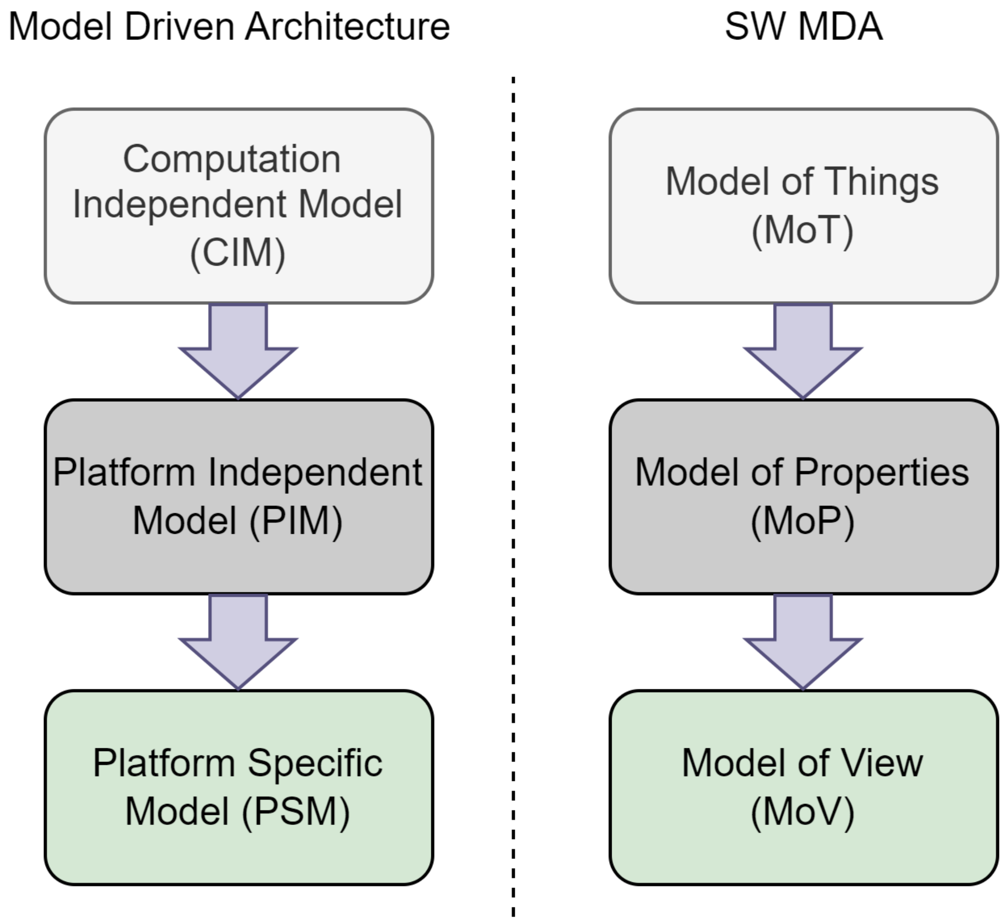
Source: D. Lettnin, M. Winterholer. Embedded Software Verification and Debugging. Springer. 2017.

Background – ISO26262-6 Standard

- ISO26262-6 specifies the requirements for product development at the software level for automotive applications
- The standard recommends the analysis of requirements and requirements based tests for all the ASIL (Automotive Safety Integrity Levels)
- To evaluate the code coverage, the standard specifies 3 metrics:
  - Statement coverage
  - Branch coverage
  - MC/DC (Modified Condition/Decision Coverage)

Background – Model Driven Arhictecture (MDA)

- MoT: Formalization of things and their intended functionality
- MoP: Abstract property model
- MoV: Final layer targeting the verification of firmware deigns



Background – Formal Verification and CBMC

- Testing-based techniques can only show the presence of bugs, not their absence
- CBMC - Bounded Model Checker for ANSI C
  - Exhaustive analysis of the code
  - Cross-function verification
  - Detection of software weaknesses
  - Branch and MC/DC coverage

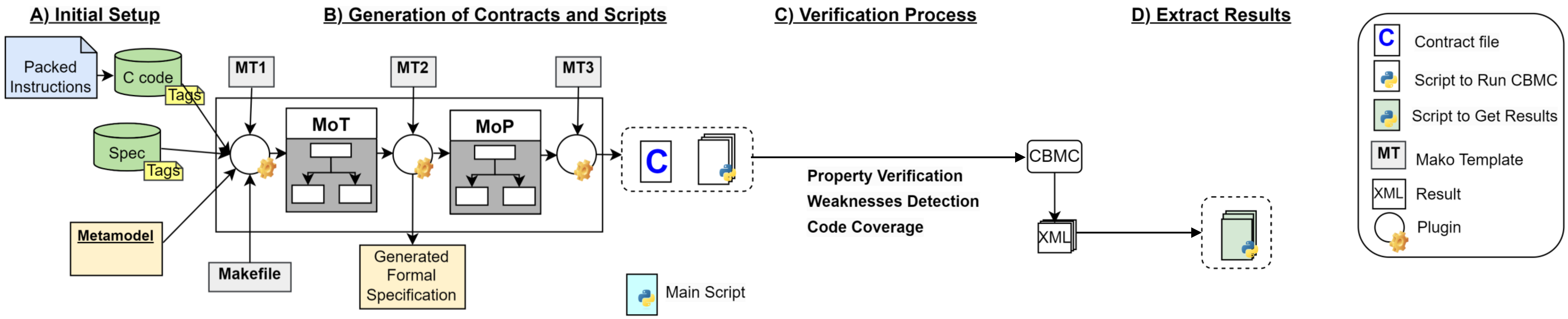
Verification Challenges - Example

```
(1) int tolerance = 8;
(2) void select_action(int v_in, int v_out){
(3)    int v_ratio;
(4)    int action;
(5)    v_ratio = abs(v_in/v_out) + tolerance;
(6)    if (v_ratio < 8){
(7)        action = 1;
(8)    } else if (v_ratio >= 8 && v_ratio <= 20){
(9)        action = 2;
(10)   } else {
(11)       action = 3;
(12)   }
(13)  if (v_in > 100 && v_out> 80){assert(action == 2);}
(14)   write_register(action);
(15) }
```

- Detection of Weaknesses
  - Line 5: Division by 0
- Unreachable paths
  - Line 6: v_ratio is never less than 8
- Safety Properties
  - Line 13: assertion must be verified for all the possible values.
- Automation of the process

```
function select_action decision/condition `v_ratio < 8' false: SATISFIED
function select_action decision/condition `v_ratio < 8' true: FAILED
```

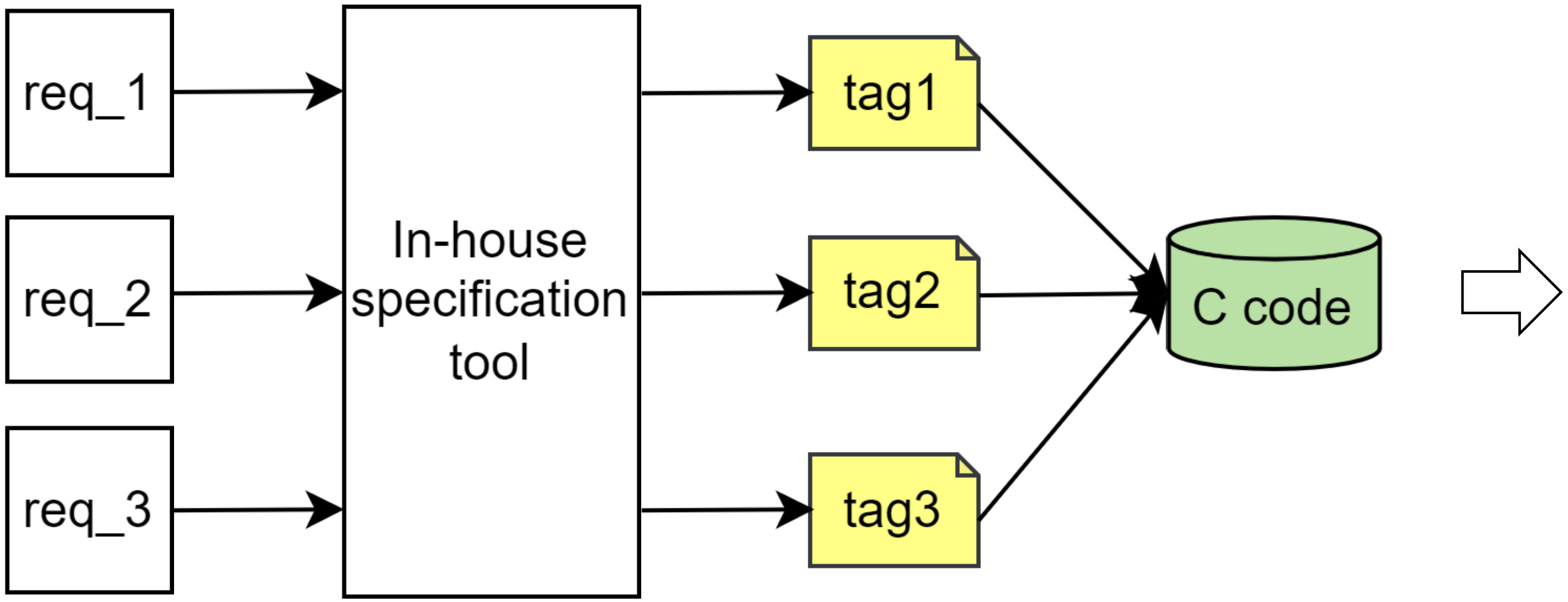
Proposed Approach - Overview



#define __mul(a,b)__ a*b

Proposed Approach – Initial Setup

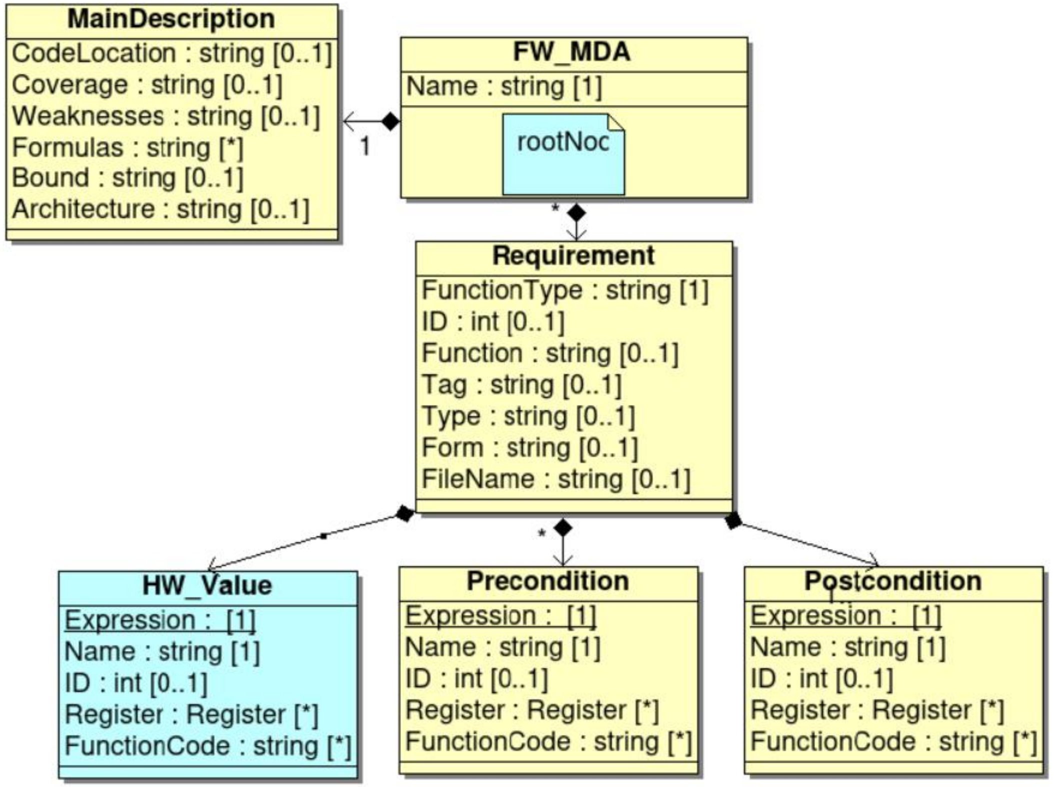
- A "tag" is generated for each requirement
- This tag is assigned to the functions of the C code




```c
void function_req_1(){
//tag1

 }

void function_req_2(){
//tag2

 }

void function_req_3(){
//tag3

 }
```

Proposed Approach – Initial Setup(2)

- The safety properties must be linked into a metamodel which includes:
  - Preconditions
  - Postconditions
  - Hardware values (boot mode, reset mode)
  - Platform parameters: bound and architecture
  - Verification parameters: type of code coverage, weaknesses under analysis



Proposed Approach – Generation of Contracts

- The contracts are generated based on the specification

```c
#ifndef FILE_DECLARATION_REQ_1
#define FILE_DECLARATION_REQ_1
#include "file_req_1.c"
#endif
#include "formulas.h"
void contract_requirement_1(){
signed int nondet_int();
      //Initial values for registers
      REGISTER_1__WRITE(nondet_int);
      REGISTER_2__WRITE(nondet_int);
      HW_VALUE(adress,value);
      // Preconditions
      if(REGISTER_1__READ() >= formula){
      // Function Under Verification
      function_req_1();
      // Postconditions
      assert(REGISTER_2__READ() == 1);
      }
      }
```

- file_req_1.c: C file implementing the requirement
- formulas.h: file with arithmetic expressions
- If required, input values in the function are also included in the contract

Proposed Approach – Generation of Contracts

- To get the main code coverage, a new contract file was generated. This file calls the functions under verification of all the contracts

```c
#include "file_contract_1.c"
#include "file_contract_2.c"
//...
#include "file_contract_n.c"
int main(){
contract_requirement_1();
contract_requirement_2();
//...
contract_requirement_n();
      }
```

accellera SYSTEMS INITIATIVE
2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Proposed Approach – Makefile Example

- The user can access the verification parameters via a Makefile

```
run_verification:
    python run_verification.py \
    --specification_file = file.xml \
    --code_folder = files/source_folder/ \
    --weaknesses=ad \
    --cc=mb \
    --bound=16 \
    --arch=32 \
    --D=__CBMC__
```

- file.xml: generated by our in-house specification tool
- source_folder: location of C code
- a (arithmetic overflow check)
- d (division by zero check)
- m (MC/DC coverage)
- b (branch coverage)
- --D: directives of the code

Proposed Approach – Generation of Scripts

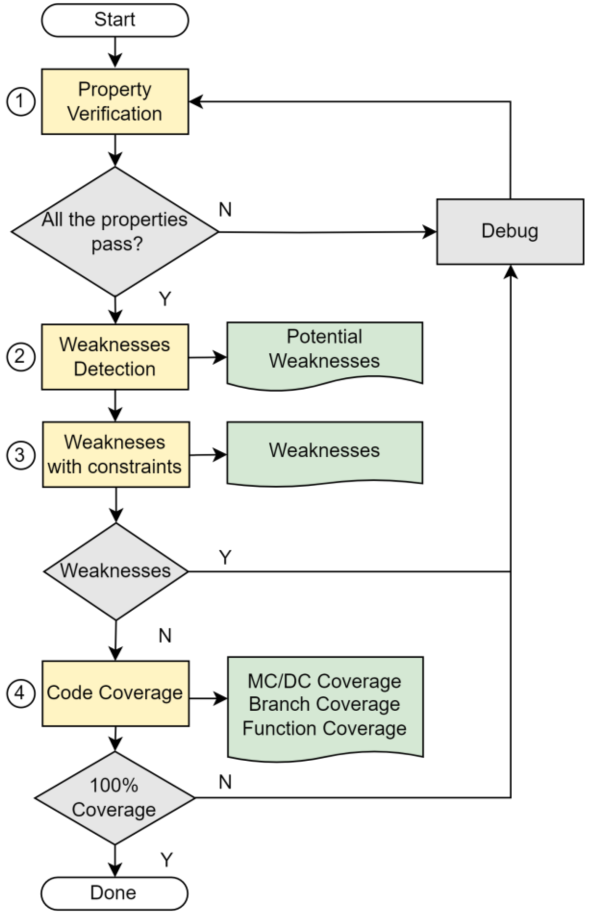
- The scripts are generated based on the Makefile

```
(file= open("results.xml",'w')
subprocess.check_call(['cbmc',
'-I', 'folder_1/',
'-I', 'folder_2/',
'-D', '__CBMC__',
'file_req_1.c',
'limits.c',
'protection.c',
'--unwind','16',
'--32',
'--cover', 'mcdc',
'--xml-ui',
'--function', 'contract_requirement_1'
],stdout=file
```

The scripts follows the syntax of CBMC:

- Include all the paths of the code folder (-I)
- Directives are added using (-D)
- Include all the C files i.e. Cross-over verification
- Include the name of the contract

Proposed Approach – Verification Process



- Verify the properties with the specified bound
- Verify the weaknesses considering the range values of the requirements
- Verify the code to check all the possible weaknesses without consider the range of values
- Compute the code coverage

Proposed Approach – Extract Results

- The output results of CBMC were filtered in order to obtain the unit coverage, file coverage and code coverage using the branch and MC/DC coverage criteria

```
2024-01-16T11:10:22.753652 VERIFICATION SUCCESSFUL
PROPERTY RUNTIME : 0 hours, 0 minutes, 0 seconds, 223 milliseconds

UNIT BRANCH COVERAGE: function_test. ** 4 of 9 covered (44.44%)
FILE BRANCH COVERAGE: file_req_1.c: ** 10 of 18 covered (55.55%)
Code coverage: ** 27 of 57 covered (47.4%)
Coverage Time: 0 hours, 0 minutes, 0 seconds, 386 milliseconds

UNIT MCDC COVERAGE: function_test. ** 9 of 22 covered (40.90%)
FILE MCDC COVERAGE : file_req_1.c: ** 21 of 37 covered (56.75%)
Code coverage: ** 44 of 109 covered (40.4%)
Coverage Time: 0 hours, 0 minutes, 0 seconds, 499 milliseconds
```

Proposed Approach – Extract Results (2)

- The output results of CBMC were filtered in order to obtain the unit coverage, file coverage and code coverage using the branch and MC/DC coverage criteria

```
TYPE:DIV-BY-ZERO-CHECK
RESULT: FAILURE
WEAKNESS TOTAL:  2
TOTAL CASES:  6
WEAKNESSES TIME: 0 hours, 0 minutes, 0 seconds, 104 milliseconds
```

- Out of 6 assertions related to division by 0, 2 have failed

```
PROPERTY: function_2.division-by-zero.2
REASON: division by zero in value / return_value_REGISTER1__GET
RESULT: FAILURE
FUNCTION: function_2
FILE: example1.c
LINE 9
```

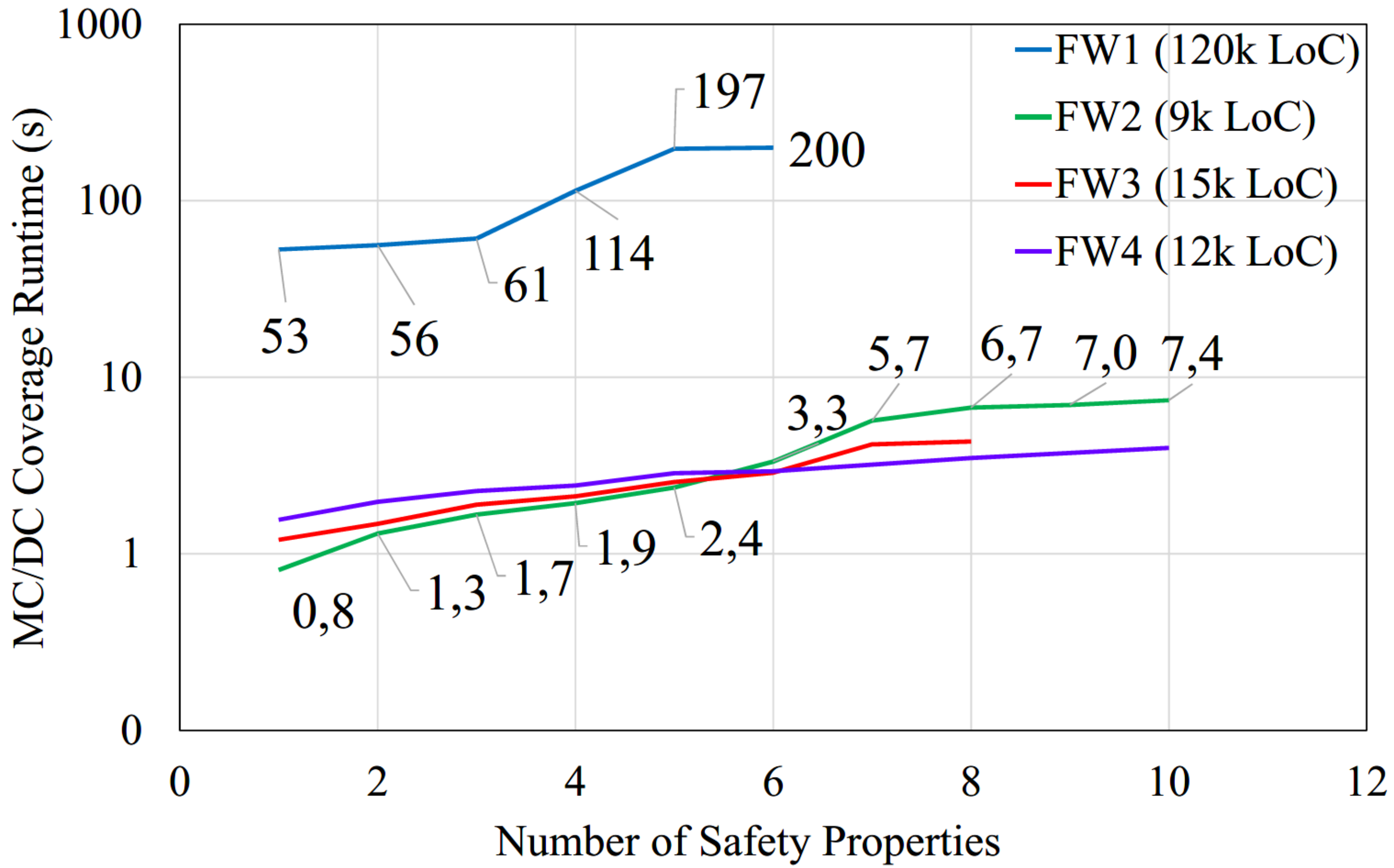
- The weakness location is printed for debugging

Results(1)

- The methodology was applied during the pre-development phase of firmware designs for the verification of safety properties

Design		Property Verification		Weaknesses Verification	Code Coverage				
Name	LoC	Safety Properties	Avg Runtime (s)	Generated assertions	Branch		MC/DC		Total Runtime
FW1	120k	6	49	9119	1.5%	2min	3,90%	3min	2h
FW2	9k	10	0,48	17675	40,60%	4s	52,20%	6s	4min
FW3	15k	8	0,675	34306	10,10%	3s	16,30%	4s	6min
FW4	12k	10	1,029	6395	11.6%	3s	13,85%	3S	4min

Results(2)



- The average runtime is determined for the complexity of the code, e.g., a code with more loops or recursive functions can be more complex even if it has less line of code.

Conclusion and Future Work

- The contracts and scripts were generated in a few seconds for all the designs
- The runtime of the properties depends on the size and complexity of the design —similar to a formal verification of hardware
- The reliability of firmware designs can be increased with the use of formal methods and the MDA

Future work:

Extend the methodology for other model checkers to automate the formal verification of concurrent designs and Rust programs

Acknowledgment

- This work has been developed in the project VE-VIDES (project label 16ME0243K) which is partly funded within the Research Programme ICT 2020 by the German Federal Ministry of Education and Research (BMBF).

Questions?

Thank you!