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

Bryan Olmos*†, Sanjana Sainath*, Wolfgang Kunz †, Djones Lettnin*
Am Campeon 1-15, Neubiberg - 85579, Germany
*Infineon Technologies AG
Email: Firstname.Lastname@infineon.com
†Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau
Email: wolfgang.kunz@rptu.de

Abstract

Electronic safety systems need to guarantee a certain level of dependability. Therefore, their development must consider not only robust hardware but also trustworthy and reliable embedded software such as firmware, drivers and bare metal software. In the case of hardware, the use of formal verification and its automation are two crucial aspects to prevent the presence of bugs during the development process. However, formal verification of firmware and especially its automation lacks a methodology that can be applied to an industrial environment. In this paper, the verification of firmware is automated to cover software criteria defined in ISO26262-6 such as requirements-based tests, code coverage and weakness detection. Additionally, it introduces the main guidelines to enable this automation in an early phase of the development. The proposed methodology is based on the Model Driven Architecture and the generation of C files —usually called contracts. These files contain the functions under verification with the preconditions and post-conditions to be verified. This work considers an end-to-end register verification, which means that the design is proved with respect to the Hardware Abstraction Layer (HAL). Furthermore, this work generates the scripts to run the open-source tool Bounded Model Checker for ANSI C (CBMC). The methodology was applied to industrial designs of 16 and 32 bits. The results show that the generator can set up the verification environment in a few seconds and similar to hardware, the verification runtime depends on the complexity of the design. The methodology can be easily adopted in industry or academic environments due to the fact that it is based on Python scripts and open-source tools.

I. Introduction

Formal verification tools have been well adopted by the industry for hardware verification. This adoption has two main reasons. First of all, other methods such as simulation can prove the limited correctness of a design but not ensure the absence of bugs. Second, the verification process has been automated successfully to reduce the effort of writing properties and set up the environment to run the formal tools. This automation allows to reduce the time-to-market since the verification process of a System on Chip (*SoC*) still consumes more than the 60% time of the overall development time [1]. In the case of high-level software, formal verification of C code has been widely studied during the last 20 years [2], [3].

The former studies have used the open-source tool CBMC [2] for the verification of properties, which describe the specification of the design under verification. These properties can be added to the C code with the use of annotations [4], [5], which describe in the right syntax of the tool, the preconditions and post-conditions of a function or a piece of code. Furthermore, it has been proved how CBMC can be used for code coverage and the detection of software weaknesses specified in the Common Weaknesses Enumeration (*CWE*) community [6].

However, the automation process for the verification of firmware is not well-defined in the industry. The main reason is that there exists a lack of comprehension about how formal methods can be applied in an industrial tool-chain to verify firmware designs based on C code.

This paper proposes to increase and scale the dependability of firmware designs through automation of the formal verification of designs based on C code following the OMG (Object Management Group) vision of Model Driven Architecture (MDA) [7], the open-source tool CBMC and a requirement based verification. This work considers a novel approach to automate the verification of firmware based on sequential C code. The main contributions of this paper are:

- A new generator is developed to automate the verification of properties, the analysis of code coverage and the detection of weaknesses.
- The generator was used to verify industrial designs that include safety-critical features.
- The methodology provides the main guidelines to be considered during the design process in order to enable automated verification.

This paper is organized as follows: In Section II, the main concepts related to software verification, functional safety, metamodeling and related work are discussed. The main challenges of firmware verification considered for this work are introduced in section III. Section IV introduces the methodology based on the Model Driven Architecture to build the generator and the guidelines for FW designs. Section V shows the experimental results. These results are discussed in section VI. The conclusions and future work are presented in Section VII.

II. Background

A. Software Verification Criteria

ISO26262-6 specifies the requirements for product development at the software level for automotive applications [8]. The standard recommends performing requirements-based tests and code coverage for all the *ASIL(Automotive Safety Integrity Levels)*. The standard specifies 3 code coverage metrics: 1) *statement coverage* measures the executable statements in the program which are invoked at least once. This coverage criterion is considered very weak and useless because it does not consider the analysis of the logic expressions [9]; 2) the *branch coverage* states that each branch direction must be traversed at least once, e.g., the condition of an if-else statement must be evaluated for true and false; 3) the *MC/DC (Modified Condition/Decision Coverage)* enhances the condition/decision coverage criterion by requiring that each basic condition be shown to independently affect the outcome of the decision [9]. Additionally, ISO26262-6 defines types of violations which can be related to memory content, e.g., buffer overflow, buffer underflow, wild pointers, memory miss-allocation and memory leaks.

B. Model-of-Things (MoT) and Model-of-Properties (MoP)

In [10], MoT and Model-of-Design (MoD) were introduced to speedup generator construction of hardware designs following the OMG (Object Management Group) vision of Model Driven Architecture (MDA) [7]. MoT and MoD are intermediate models for hardware design based on metamodeling and code generation. In [11], the MoP was integrated to the the MDA to allow the generation of properties for an Assertion Based Verification (ABV). In these methodologies, the workflow begins with the capture of the design specification into the MoT. Afterwards, the MoT is transformed into the MoP which represents the abstract property model and is platform-independent; and the MoP was reused to generate code for different platforms. In this project, MoT and MoP are used for metamodeling the requirements and the generation of contracts for C code.

C. Related Work

There are previous works in which authors proposed the verification of firmware. In [12], for example, authors verify it using firmware-hardware interaction patterns, but without considering simulation or automation. New techniques for FW development and verification are introduced in [13]. However, they consider a firmware design based on Assembly code, similar to the work in [14]. For this reason, they are not suitable for our purpose. [15] proposes the use of inline functions during the design phase and the reuse of these functions for verification. We also consider this idea for our methodology. However, the previous approach does not consider formal methods or automation. In [6], the authors verify software weaknesses. Nevertheless, the process to set up CBMC is manual and the source code is not focused on FW designs. Concerning the use of contracts and annotations for the verification of C functions are multiple works, for example, [5] uses contracts for the verification of high-level C code. Additionally, [16], [17] use VCC (Verified C compiler) which translates C program with annotations to an intermediate language before its verification; [18] uses comments as annotations for the C program. In the cited works, they use annotations in the source code.

III. Firmware Challenges Under Verification

The verification of the firmware has different challenges during its development. In this work, we consider 3 main challenges.

A. Unreachable Paths

The code coverage helps to test the effectiveness of our testing, which means, it is a metric to measure the percentage of executed code using our test cases. It is not always feasible to reach 100% of code coverage without a detailed analysis of the code or the set of test cases. One reason is the presence of unreachable paths. This issue becomes more relevant in firmware designs due to the fact that the execution of the code not depends only on the software, but also on the values and interaction of the hardware registers, for example, considering the next typical situations.

First situation, a safety condition for over-voltage was implemented in a 16-bit data width microcontroller, as shown in Fig. 1a). The safety condition must be TRUE if the voltage is higher than a constant called MAX_VALUE (line 5). Thus, the SW developer provides an initial MAX_VALUE of 65534 (line 1), so the safety condition can be triggered with a value of 65535. However, the path to reach the condition can be unreachable in case the READ_VOLTAGE() function casts the value or does not consider all the bits of the register — intentionally or unintentionally. In these situations, none test will cover the condition and we can detect these unreachable paths only after running the test cases and analyzing the code.

```
(1) #define MAX_VALUE 65534
(2) const uint16_t voltage;
(3) void function_test(){
(4)
(5)  voltage = READ_VOLTAGE();
(6)  if (voltage > MAX_VALUE){
(7)     condition(TRUE);
(8)  } else {
(9)       condition(FALSE);
(10)  }
(11) }
```

```
(1) void function_enable()
(2) {
(3) while(READ_COUNTER()!=5);
(4) function_init();
(5) #ifdef __CBMC__
(6) //Syntax CBMC
(7) signed int nondet_int();
(8) WRITE_COUNTER(nondet_int());
(9) #endif
(10) while (READ_COUNTER()< 10);
(11) enable(TRUE);
(12) }
```

```
file= open("results.xml",'w')
subprocess.check_call(['cbmc',
'-I', 'folder_1/',
'-I', 'folder_2',
'-D', '__CBMC__',
'file_1.c',
'file_2.c',
'--unwind','10',
'--32',
'--cover', 'mcdc',
'--xml-ui',
'--function', 'function_test'
],stdout=file
```

Fig. 1: a) Example Over-voltage b) Example of a sequence of events c) Script Sample

Second situation, to enable the operation of a microcontroller it needs sequence of events of a register called COUNTER, which can only be modified by the HW. The implementation of this sequence of events code is shown in Fig. 1b). The first while loop (line 3) will end only if the value is equal to 5 and the second while loop (line 10) ends only if the value is higher or equal to 10, these 2 conditions cannot be reached for any initial COUNTER value and the enable condition will not be reached in a software simulation. Additionally, after the first while loop, a function_init() is called (line 5), which can also be affected by the value of the register COUNTER.

In these 2 examples, unreachable paths will affect the verification. Traditionally, they can be identified after writing, running and analyzing the code coverage of a set of test cases. To overcome this challenge, in this work, we propose the code coverage in an initial phase using formal verification, for example, we can invoke function_test() or function_enable() using a contract without constraints and run CBMC using a Python script, as shown in 1c). It will return the code coverage. In case we detect unreachable paths due to sequences of events, we can add directives with the syntax of CBMC such that we verify the HW registers for all the possible values, as shown in 1b).

B. Safety Properties

Informally, safety properties stipulate that “bad things” do not happen during program computations [19]. In this work, we consider “bad things” the hazards due to a malfunctioning behavior of the firmware. ISO26262-1 defines a hazard as a potential source of harm caused by the malfunctioning behavior of a system [8]. These safety properties can be implemented as functions in the C code that monitor the sensors of the embedded system and prevent harm. The functions related to safety-critical designs are susceptible to functional faults for example: operating faults: omission or unnecessary operations; conditional faults: incorrect condition or limit values; behavioral faults: incorrect behavior, not conforming to requirements [20]. The verification of these conditions using simulation techniques becomes unfeasible when the number of registers implementing these functions or their bit length increases, for example, consider a property for a sensor that contains two channels to measure voltages and the requirements are:

- Preconditions: Voltage1: [0, 65535], Voltage2: [0, 65535], Configuration Parameter: [0,127].
- Postcondition: Set error bit if the temperature is above limit_1

The value of limit_1 is dynamic and depends on Voltage1, Voltage2 and the Configuration Parameter. To test if the error is always set to 1, we have to consider all the combinations of these values, that is, 549 755 813 888 (65536*65536*128) tests. To overcome this challenge, we propose the use of formal verification, which has been largely analyzed for high-level application code using formal methods and the previous studies show that the runtime can be reduced to milliseconds.

C. Adoption of Formal Verification in the Industry

The use of formal tools in the industry has increased in the last years, especially for friendly hardware designs such as interrupt controllers, bus protocol bridges, controllers and crossbars [21]. However, the formal verification firmware is still not yet defined and requires not only some fundamental concepts about the field but also training and expertise. To overcome this challenge, we propose an automated workflow.

IV. Propoposed Approach

Fig. 2 shows the four steps considered in the methodology. All these steps are automated in this work and the parameters for the verification can be accessed by the user using a Makefile.

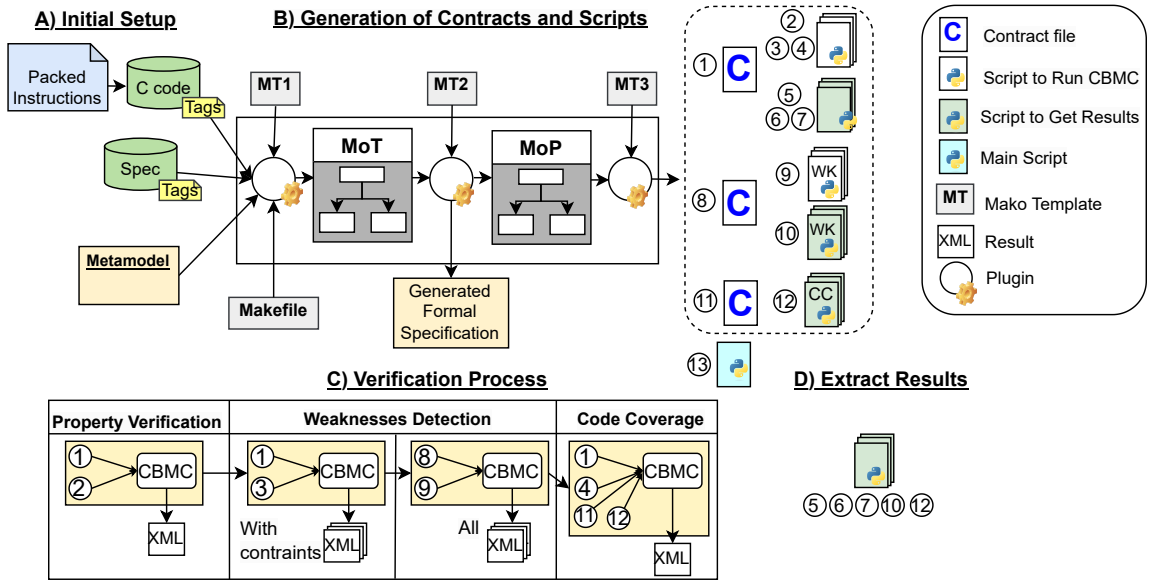

Fig. 2: Proposed Approach for Automating the Formal Verification of Firmware

A. Initial Setup

In the initial phase, the requirements of the safety properties must be linked into a metamodel which includes the preconditions, postconditions, hardware values and the parameters for the verification such as bound and architecture. In this methodology, the user must manually fill this initial information in our in-house verification specification tool. The estimated effort for a user is 10 minutes by requirement. It is also necessary to have a link between the source code and the requirements. Thus, each time the user adds a requirement an alphanumeric string called "tag" is generated for our in-house verification specification tool. This tag is assigned to the body of the functions of the C code as a comment, this tagging process can be done during the generation of the skeleton of the C code or it can be added manually. Additionally, it is recommended the use of inline functions which access the special function registers (SFRs) and contained bitfields. These functions can automatically be generated using metamodel techniques and they help for the early start of firmware development and verification [15]. In case these SFRs are not defined, the address and the values of the registers can be specified manually in the specification.

In this step, we need to add an additional file with the definition of the packed instructions of the compiler or specific platform, for example, the instruction $mul(a, b)$ can be defined as #define mul(a,b) $a * b$.

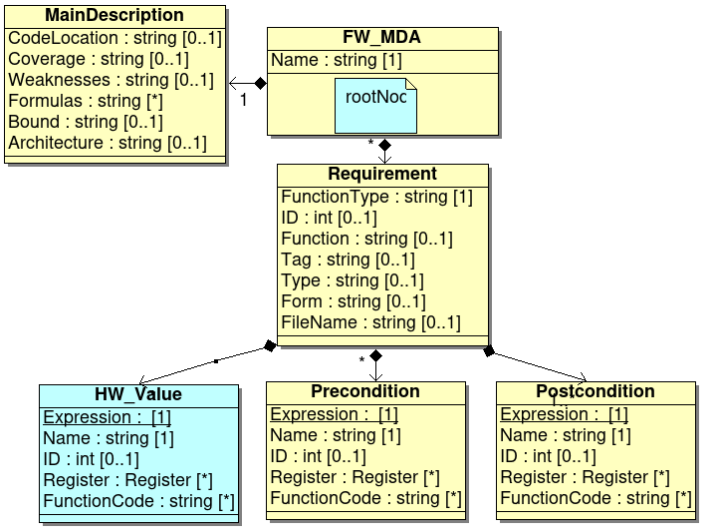

Fig. 3: MoT Metamodel for FW verification (simplified)

B. Generation of Contracts and Scripts

The generation of these contracts and the scripts —to run CBMC for property verification, weaknesses detection and code coverage —is based on intermediate models called MoT and MoP. This generation is based on three Mako templates which define the syntax of the generated files [22]; and three plugins, which are written in Python language, as shown in Fig. 2. A simplified version of the MoT is shown in Fig. 3. It consists of a root node and components to describe the preconditions, postconditions and HW values —such as boot mode or reset mode —of each requirement. It also includes the option to run the tool such as the type of code coverage and the weaknesses to detect.

In this work, the preconditions or postconditions are not written in the source code. This is because it can introduce errors during the verification process. This work proposes the use of contracts, as shown in Fig. 4a). They contain all the information such as initial values (lines 9 - 10), preconditions (line 13), FUV (Function Under Verification) (line 15) and postconditions (line 17). It needs to include the C file which contains the FUV as shown in line 3 to call the function and its elements. In order to avoid a situation in which the file is declared multiple times for different requirements accessing the same C file and causing compilation errors, it was necessary the use of directives as shown in line 1. The C file is added to the compiler only in the case that it was not added before. It is useful in a situation where multiple contracts are called for verification, e.g., in the contract file of the main code coverage.

In this example, the FUV is called with the constraints because CBMC will check if the bound is enough to consider all the states, and calling the function without constraints will cause this check to fail —depending on the design. This is because, BMC (Bounded Model Checking) is applied by CBMC, which can determine if the bound k is enough to consider all the states of a C program or not. This bound can be specified in a Makefile which contains the parameters to run the verification. In the case of firmware, the bound is well-known during the design. This is because it has to request a task to the hardware and wait until it is completed. In order to prevent a scenario where the system is stuck waiting for the hardware, it is necessary to add HW values (line 11). These HW values are also part of the specification, for example, a boot mode or reset mode expected for a requirement.

The scripts to run the weaknesses consider all the in-built assertions of CBMC. They are specified in the Table I and could be verified using one single script, as shown in 1c). However, it makes the debugging process very hard. Additionally, the user can decide to only verify the code for certain weaknesses. For this reason, one script was generated for each weakness depending on the parameters specified for the user in a Makefile. The nomenclature for the Makefile is presented in Table I. It has to be done for each requirement in case the code size makes not possible to verify all the code at once.

To get the main code coverage, a new contract file was generated. This file calls the functions under verification of all the contracts as shown in Fig. 4b). The generated files shown in Fig. 2 are:

1) Contract file with constraints: It contains the FUV, the preconditions, the postconditions and initial values.
2) Script for property verification: It runs CBMC and includes the location of the C files, header files, name of the function under verification, bound and options to generate an XML file with the results, as shown in 1c).

```
(1)#ifndef FILE_DECLARATION_REQ_1
(2)#define FILE_DECLARATION_REQ_1
(3)#include "file_req_1.c"
(4)#endif
(5)#include "formulas.h"
(6)void contract_requirement_1(){
(7)signed int nondet_int();
(8)       //Initial values for registers
(9)       REGISTER_1__WRITE(nondet_int);
(10)      REGISTER_2__WRITE(nondet_int);
(11)      HW_VALUE(adress,value);
(12)      // Preconditions
(13)      if(REGISTER_1__READ() >= formula){
(14)      // Function Under Verification
(15)      function_requirement_1();
(16)      // Postconditions
(17)      assert(REGISTER_2__READ() == 1);
(18)      }
(19)      }
```

```
(1)#include "file_contract_1.c"
(2)#include "file_contract_2.c"
(3)//...
(4)#include "file_contract_n.c"
(5)int main(){
(6)contract_requirement_1();
(7)contract_requirement_2();
(8)//...
(9)contract_requirement_n();
(10)      }
```

Fig. 4: a) Proposed contract for FW Verification b) Proposed contract for code coverage with n requirements

TABLE I: CWEs detected by CBMC

Weakness	CBMC command	Makefile	Example of assertions[a]
Buffer Overflow	–bounds-check	b	`while(i<size){i++; assert(i < size);}`
Pointer Check	–pointer-check	p	`ptr = &val; assert(ptr != NULL);`
Pointer Overflow	–pointer-overflow-check	o	`int arr[SIZE]; int* ptr = &arr[0]; ptr += SIZE; assert(ptr >= &arr[0] && ptr < &arr[SIZE]);`
Memory Leaks	–memory-leak-check	m	`int *p = malloc(sizeof(int)); assert(p != NULL);`
Division By Zero	–div-by-zero-check	d	`c=a/b;assert(b != 0);`
Arithmetic Overflow	–signed-overflow-check	s	`uint16_t a,b,c; c= a+b; assert(!(a+b >= 32768));`
Floating Overflow	–float-overflow-check	f	`float a,b,c; c = a + b; assert(!isinf(c));`
Conversion Error	–conversion-check	c	`(uint16_t)(a+b);assert(!(a+b >= 32768));`
Undefined shifts	–undefined-shift-check	t	`uint32_t x, y, n; x = y << n; assert(n <= 31);`
Not a number	–nan-check	n	`double x = sqrt(-1);assert(is_nan(x));`
Code Coverage	**CBMC command**	**Makefile**	**Description**
MC/DC Coverage	–cover mcdc	m	It states that each branch direction must be traversed at least once
Branch Coverage	–cover branch	b	Each basic condition in a decision must affect the outcome of the decision

[a] These examples illustrate the usage of assertions to detect weaknesses, but CBMC has its own set of functions as detailed in [23]

3) Scripts for weaknesses detection: Python scripts to run CBMC and detect the software weaknesses specified in the Makefile. The options are specified in Table I
4) Scripts for code coverage: Python scripts to run CBMC and return the results of the code coverage for each function under verification in an XML file.
5) Scripts to get the main results of the property verification.
6) Scripts to get the main results of the weaknesses detection.
7) Scripts to get the main results of the code coverage.
8) Contract file without constraints.
9) Script to run the contract files without constraints.
10) Scripts to get the main results of the weaknesses detection without constraints.
11) Contract file for code coverage: It calls all the contracts for function verification as shown in Fig. 4a).
12) Script to run CBMC and return the overall result for code coverage in an XML file.
13) Main script to execute all the scripts and summarizes the results for each requirement.

The user must add the input parameters for weaknesses detection and code coverage into a Makefile using, for example, the format weaknesses= "d" (division by zero and pointer check) and coverage = "m" (MC/DC coverage). The number of generated files N can be calculated as $N = 4 + n \times (5 + 6 \times w + 3 \times c) + 3 \times c$, where n is the number of requirements, w is the number of weaknesses under verification and c is the code coverage parameters selected by the users.

C. Verification Process

Fig. 2 shows the verification workflow considered for the implementation. After the analysis of unreachable paths, CBMC checks that the code meets the requirements with the specified bound; second, CBMC checks the weaknesses taking into account the range values of the requirements; third; it verifies the code to check all the possible weaknesses without consider the range of values —in this step is also verified if the bound is enough for all the possible values; finally, the code coverage is computed.

D. Extract Results

The results of CBMC were filtered in order to obtain the unit coverage, file coverage and code coverage using the MC/DC and branch coverage criteria. It was necessary to filter the results because depending on the sizes of the code, the XML files can contain easily hundreds of thousands of lines, for example, the XML file with the results for code coverage of the firmware 1 has 900k lines (80MB).

V. Experimental Results

The methodology was applied to firmware designs of 16 and 32 bits. As shown in Table II, the contracts and scripts were generated in a few seconds for all the designs. The analysis considers the effective LoC (Lines of Code). The runtime of the properties depends on the size and complexity of the design —similar to a formal verification of hardware. One bug was detected during the pre-development phase of the FW2. It was due to a miss bound considered during the design phase. It was not detected running regression simulations during 24 hours and the results show that it was a corner case that will give a successful result in 99.99739% of the tests. CBMC was able to generate 17675 assertions for the FW2 and it can detect 245 potential weaknesses and identify 21 false positives. In this case, more weaknesses were detected in the FW1 with fewer lines of code than FW2, FW3 and FW4. This is because, it implements arithmetic operations that are sensitive to overflow and conversion error issues.

TABLE II: Main Results for Industrial FW Designs

Design		Property Verification		Weaknesses Verification			Code Coverage				
Name	LoC	Safety Properties	Avg. Runtime	Generated assertions	With constraints	Without constraints	Branch		MC/DC		Total Runtime
FW1	120k	6	49 s	9119	14	2	1.5%	3min	3,90%	3min	2h
FW2	9k	10	0,48 s	17675	245	21	40,60%	4s	52,20%	6s	4min
FW3	15k	8	0.675 s	34306	174	0	10,10%	3s	16,30%	4s	6min
FW4	12k	10	1.029 s	6395	0	0	11,6%	3s	13,8%	3s	4min

As shown in the Fig. 5, the runtime of the code coverage increases linearly with the number of added requirements.

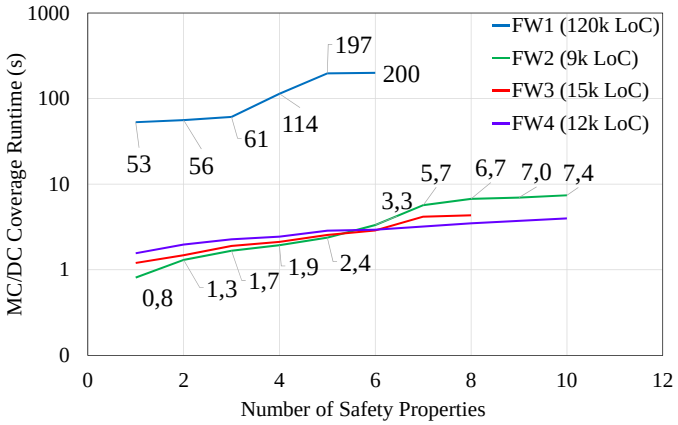

Fig. 5: Runtime for code coverage MC/DC

VI. Discussion

The runtime of the code coverage depends on the size of the code and the number of properties. Not all the code was verified for the automation because this work was focused on the safety properties of the design. However, it was possible to determine the branch and MC/DC coverage as was proposed in the introduction. The properties

of the FW1 were verified using contracts with an average runtime of 49s, the same verification was not feasible using annotations directly in the code due to its extension. This shows the adaptability of the proposed approach. The verification setup was generated quickly for all the designs using the proposed approach. Consequently, the generated contracts can be reused by other Model Checkers, and the scripts can be adapted with the modification of the Mako templates.

VII. Conclusion

In this paper, we have presented a framework that automatically generates the setup for the formal verification of firmware designs based on C code. During the experiments we detected bugs and weaknesses that were not found using simulation and emulation. We can conclude that the reliability of firmware designs can be increased with the use of formal methods and that the process can be automated in an industrial environment using the Model Driven Architecture. Additionally, we show how the code coverage can be used in an initial phase to avoid unreachable paths. As the next step, we plan to extend the methodology for other model checkers in order to automate the formal verification of concurrent designs and programs in embedded systems that are based on the Rust programming language [24].

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). Furthermore, we thank to our colleagues Endri Kaja, Johannes Grinschgl, Ravi Marathe and Basavaraj Naik for their constant feedback and support.

References

[1] W. R. Group, "2022 Wilson Research Group Functional Verification Study: FPGA Functional Verification Trend Report," Siemens, White Paper, 2022.
[2] "Bounded Model Checking for Software," http://www.cprover.org/cbmc/, accessed: 2023-07-6.
[3] E. Clarke, D. Kroening, and F. Lerda, "A tool for checking ANSI-C programs," *Lecture Notes in Computer Science*, vol. 2988, pp. 168–176, 01 2004.
[4] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte, "VCC: Contract-based modular verification of concurrent C," in *2009 31st International Conference on Software Engineering - Companion Volume*. Vancouver, BC, Canada: IEEE, 2009, p. 429–430.
[5] N. Chong, B. Cook, J. Eidelman, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow at amazon web services," 2021.
[6] M. Byun, Y. Lee, and J.-Y. Choi, "Analysis of software weakness detection of CBMC based on CWE," in *2020 22nd International Conference on Advanced Communication Technology (ICACT)*, 2020, pp. 171–175.
[7] "Model Driven Architecture," https://www.omg.org/mda/, accessed: 2023-06-13.
[8] "IS026262 Road vehicles – Functional safety, Part 1: Vocabulary, Part 6: Product development at the software level." International Organization for Standardization (ISO), Standard, 2026.
[9] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," Tech. Rep., 2001.
[10] W. Ecker and J. Schreiner, "Introducing Model-of-Things (MoT) and Model-of-Design (MoD) for simpler and more efficient hardware generators," in *2016 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 2016, pp. 1–6.
[11] K. Devarajegowda and W. Ecker, "On generation of properties from specification," in *2017 IEEE International High Level Design Validation and Test Workshop (HLDVT)*, 2017, pp. 95–98.
[12] S. Ahn and S. Malik, "Automated firmware testing using firmware-hardware interaction patterns," in *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. New Delhi India: ACM, Oct 2014, p. 1–10.
[13] V. Herdt, W. Kunz, D. Grose, R. Drechsler, C. Gerum, A. Jung, J.-J. Benz, O. Bringmann, M. Schwarz, and D. Stoffel, "Systematic RISC-V based firmware design," in *2019 Forum for Specification and Design Languages (FDL)*. Southampton, United Kingdom: IEEE, Sep 2019, p. 1–8.
[14] C. Villarraga, B. Schmidt, J. Bormann, C. Bartsch, D. Stoffel, and W. Kunz, "An equivalence checker for hardware-dependent embedded system software," in *2013 Eleventh ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE 2013)*, 2013, pp. 119–128.
[15] V. B. Kleeberger, S. Rutkowski, and R. Coppens, "Design & verification of automotive SoC firmware," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2015, pp. 1–6.
[16] E. Cohen, M. A. Hillebrand, S. Tobies, M. Moskal, and W. Schulte, "Verifying C programs: A VCC tutorial," p. 26.
[17] H. Liang, D. Zhang, X. Pei, X. Jia, G. Li, and J. Xu, "A correctness verification method for c programs based on VCC," in *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing (CSCloud)*. Beijing, China: IEEE, Jun 2016, p. 172–177.
[18] L. Sriya, A. M. Manohar, and N. Kumar, "Verification of C programs using annotations," in *2019 IEEE Tenth International Conference on Technology for Education (T4E)*. Goa, India: IEEE, Dec 2019, p. 106–109.
[19] B. Alpern and F. Schneider, "Recognizing safety and liveness," *Distributed Computing*, vol. 2, pp. 117–126, 09 1987.
[20] R. Lutz, "Analyzing software requirements errors in safety-critical, embedded systems," in *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, 1993, pp. 126–133.
[21] K. Devarajegowda, L. Servadei, Z. Han, M. Werner, and W. Ecker, "Formal verification methodology in an industrial setup," in *2019 22nd Euromicro Conference on Digital System Design (DSD)*, 2019, pp. 610–614.
[22] "Mako Templates," https://www.makotemplates.org/, accessed: December 29, 2023.
[23] "CBMC Documentation," https://markrtuttle.github.io/cbmc-documentation/index.html, accessed: December 31, 2023.
[24] "The Rust Programming Language," https://www.rust-lang.org/, accessed: December 29, 2023.