Doctoral Dissertation

PROGRAM VERIFICATION FOR MICROCONTROLLER-SPECIFIC CODING RULES

Nguyen Thi Thuy

Supervisor Professor Toshiaki Aoki

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

March 2023

Abstract

Microcontrollers are small computers specially designed for embedded systems. Since they were introduced, microcontrollers have been used in a wide range of applications, including mission-critical applications such as satellite rockets, automotive systems, and medical devices. Hence, it is essential to ensure the reliability of microcontroller-based applications. When developing these applications, hardware-dependent features must be considered in addition to standard C language features. For hardware-dependent features, a hardware manual is provided for each microcontroller. In this manual, information that requires special attention as coding rules are emphasized, along with the explanation of the features of the hardware. Currently, the process of verifying these coding rules is performed manually, as no single tool can directly handle this task. The reason is that the coding rules are non-standard while existing verification tools often support standard coding rules only. Verifying the coding rules is time-consuming and laborious as both the volume of the embedded source code (e.g., thousands of lines of code) and the number of coding rules are often large (e.g., the hardware manual of a popular microcontroller contains 2415 pages and 492 fragments that describe notes or cautions).

Several researchers tried to handle this task. In general, they tended to create new tools or extend existing tools. Implementing new tools or extensions is a heavy and inflexible solution. Additionally, new hardware models are frequently introduced. It is impractical to introduce one verification tool for each model.

This research aims to automate these coding rules' verification processes by proposing a flexible approach. Specifically, we proposed a verification framework that utilizes advanced techniques in program analysis and model-driven engineering. Firstly, the program analysis techniques (i.e., pattern matching, abstract interpretation-based static program analysis, bounded model checking, and counterexample-guided abstract refinement) are combined to analyze C programs effectively. Secondly, heuristic-based natural language processing techniques are employed to analyze the hardware manual and extract hardware knowledge. Thirdly, model-driven engineering techniques are employed for comprehensively modeling the hardware, compiler, and source code of microcontroller-based systems. Finally, model querying techniques enable flexibly verifying the system against the target coding rules.

The approach was evaluated by applying to handle a benchmark source code and an industrial source code. The experiment with the benchmark source code showed that the approach is feasible in verifying microcontroller-based systems against the register-access coding rules. Although the benchmark was a small-size

source code only, the source code represented different ways for violations to occur. The approach analyzed the source code successfully and detected all expected violations of the target register-access coding rules. We even find violations that senior developers miss. The experiment with the industrial source code showed that our verification framework was applicable to a real product. The precision in this experiment was 0.8, as two false warnings were detected. The recall was 1, as all expected violations were found.

This research contributes a practical solution for an important problem in the industry. Formal verification theory is highly established through a long history of development. Many methods proposed in academics could work well with small benchmark source code, but not many are applicable to industrial applications. The industrial setting is much more complex in comparison with laboratory environments. Among needs in the industry, verifying systems against microcontroller-specific coding rules is a heavy yet critical task. There is an emergency call for automated solutions. In response, our work is expected to be a practical and effective solution to judge the conformance of the coding rules and a tool for other tasks like source code understanding. The proposed verification framework promises to reduce the huge amount of manual tasks in the current verification process in practice.

Keywords: Microcontroller-specific coding rules, C programs, Program analysis, Knowledge modeling, Program verification

Acknowledgment

Firstly, I would like to sincerely thank my supervisor - professor Toshiaki Aoki. While conducting this research, professor Aoki gave me a lot of helpful advice about research and attitude when facing difficulties. He has inspired me to explore new things as well as patiently taught me to be strong and self-confident. He has also encouraged me to enjoy life by suggesting a better experience in Japan.

I would like to thank my second supervisor - senior lecturer Takashi Tomita, my advisor for minor research - professor Nguyen Le Minh, and assistant professor Tatsuji Kawai for their enthusiastic support during the time I conducted this research. I also deeply appreciate the support of Aoki laboratory's members. Some parts of this research are inspired by their discussions.

I would like to thank professor Hidehiko Masuhara of the Tokyo Institute of Technology, associate professor Takashi Ishio of the Nara Institute of Science and Technology, professor Kiyofumi Tanaka, and associate professor Daisuke Ishii for useful discussions and comments on this dissertation.

I would like to thank Aisin Software Co., Ltd.. and the company's members for their financial support and helpful advice when I conducted this research.

Finally, I would like to spend some words with my family and friends. This dissertation could not have been completed without their encouragement and support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Background

A microcontroller is a small computer usually containing one or more CPUs with memory and input/output peripherals. Microcontrollers were introduced in the 1970s as cost-efficient for embedded systems. Nowadays, microcontrollers have become popular in a wide range of electronic devices. We can easily find microcontrollers in a wide range of electrical systems, from simple systems, such as children's toys or remote control, to safety-critical systems, such as medical devices [2] and automotive systems [3]. As microcontrollers are used in safety-critical systems, the reliability of microcontroller-based systems is crucial. We may accept that the remote control in our house is broken with a little uncomfortable feeling, but we never want to sit with software bugs in a car or even live with faulty software implanted in our bodies.

Historically, there have been noticeable bugs in the embedded system with serious consequences, such as the Ariane 5 disaster in 1996 [4], or the recall of Toyota Prius [5]. Additionally, as manufacturers can only update some systems, faulty systems can become nearly useless due to the high cost of recalling and updating at the factory. Hence, we need to ensure the safety of the systems written on top of the hardware besides the reliability of the hardware itself.

Each microcontroller usually comes with a hardware manual for specifying the correct usage of the hardware. This manual contains explanations of the microcontroller's features. Along with each feature, important requirements are emphasized to require special attention from users. In this research, we call these requirements as microcontroller-specific coding rules.

To develop microcontroller-based systems, we must consider standard programming language features and microcontroller-specific coding rules. For instance, assume we have a microcontroller with two registers named `register_1`

```
1
2  void main(void) {
3      unsigned long* register_1 = (unsigned long*) 0
           xFFE20244;
4      unsigned long* register_2 = (unsigned long*) 0
           xFFE51094;
5      unsigned long val = 0x11001100;
6      unsigned long mask = 0x4;
7      *register_1 = val;
8      if (*register_1 & mask != 0) {
9          *register_2 =  0x11001100;
10     }
11 }
```

Figure 1.1: Example microcontroller-based source code

and `register_2`; the addresses of these two registers are `0xFFE20244` and `0xFFE51094`; there is a coding rule which requires that *"The written value of the two registers must not be* `0x11001100` *at the same time."*. Figure 1.1 shows a sample microcontroller-based source code written on top of the hardware. There is a violation of the coding rule as at lines 7 and 9, `register_1` and `register_2` are written to `0x11001100`, respectively.

Several works [6, 7, 8] have proposed to solve this problem. In [7], Schlich et al. created a tool for verifying the assembly code of a microcontroller by applying model checking. In [6] and [8], existing static analysis tools were extended for microcontroller-specific coding rules. In general, existing works introduced new or extended tools to handle several microcontrollers' coding rules. However, these works were hard to be extended for other coding rules. To handle other rules, it is necessary to modify the tools directly. Additionally, different microcontrollers may have different features. Sometimes, we must rewrite the tool or the extension for nearly every microcontroller. Adjusting the tool/extension to adapt to a new microcontroller model requires professional skills in both the microcontroller and the verification tool.

1.2 Research problem and objective

Currently, verifying those coding rules is an entirely manual process. This manual process makes the verification phase the most time-consuming phase in the development process. Additionally, as microcontroller-based systems are often large

and complex, human inspection is inadequate to ensure the absence of defects. Automated tools are expected to be a great support for the development process. Companies like Aisin Software Co., Ltd (AISW) are seeking automatic solutions. However, currently, there is no industry-strength tool to handle this problem.

In this research, we aim to automate the process of verifying the microcontroller-based systems against the microcontroller-specific coding rules. Multiple sources of information are needed for this verification task, including the information on the microcontroller, the behaviors of the target source code, and the used compiler. The information about a microcontroller can be found in its hardware manual, as discussed previously. The information on the behaviors of the target source code can be obtained by analyzing the semantics of the source code. Several settings of the used compiler, such as pairs of types and data sizes, can be found in the compiler manual.

Assuming that the settings of the used compiler are provided, we propose to extract, model, and use the hardware and source code information for the verification objective. To achieve this objective, there are four tasks to be solved. The first task is to analyze the hardware manual to get information about the coding rules. The main difficulty in analyzing the hardware manual is the ambiguity of natural language. The second task is to analyze the target source code and extract information about related code fragments to the coding rules. The difficulty in analyzing source code is the increasing complexity of embedded systems. The third task is to model the extracted information from the two sources above for the verification task. The difficulty of modeling the extracted knowledge is that the knowledge size is expected to be large, and the relations among the knowledge components are complex. The fourth task is to effectively use the represented information to verify the target source code against the coding rules. The difficulty of verifying the systems against the target coding rules is the huge number of microcontroller-specific coding rules.

1.3 Proposed solution

Figure 1.2 shows the overview of the proposed framework to solve the four tasks mentioned above. This approach has four phases corresponding with solutions for the tasks. The solutions to these tasks are illustrated in the four boxes named *Hardware Knowledge Extraction*, *Software Knowledge Extraction*, *Knowledge Modeling*, and *Knowledge Querying*.

Hardware Knowledge Extraction is a phase to extract information from the hardware manual. The input of this phase is a *Hardware manual* in Portable Document Format (PDF) format. The output is the *Formalized hardware knowledge* extracted from this manual. For formally describing the hardware knowledge, we

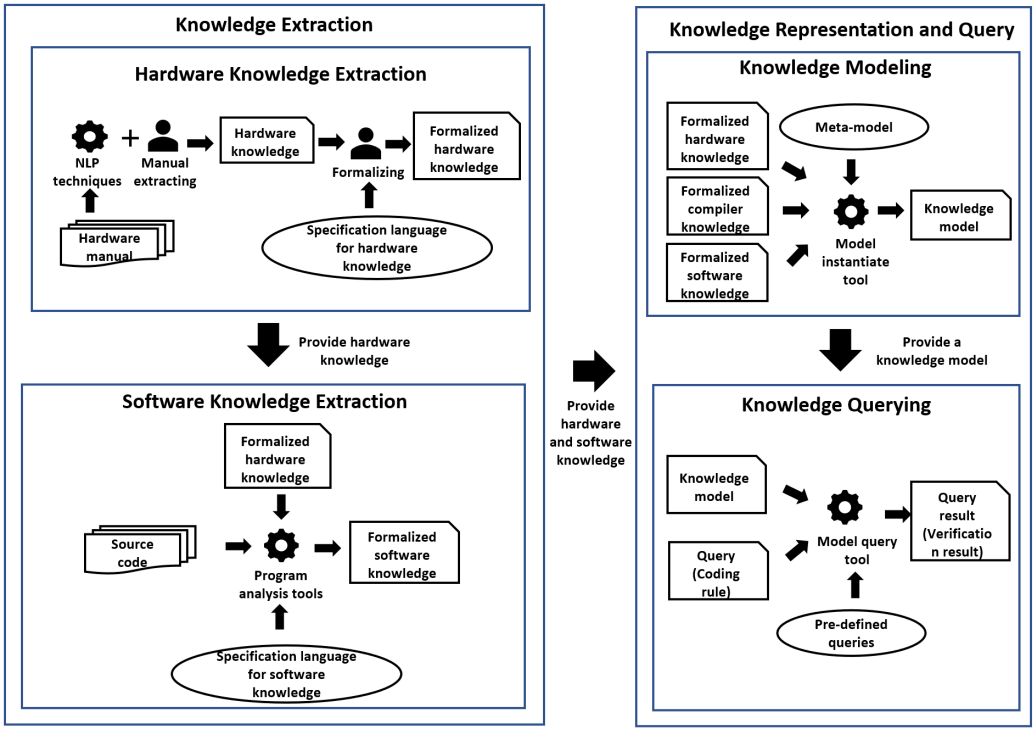

Figure 1.2: Overview of proposed framework

propose *Specification language of hardware knowledge*. Detail of this specification language can be found in Section 3.3. In the current verification process, developers read the manual and extract the coding rules based on several special keywords such as "note" or "caution". This research partially automates this process. We apply heuristics based on keywords and the documents' syntactic features to extract the coding rules' semantics. Details of the approach for handling hardware manuals can be found in Chapter 3.

Software Knowledge Extraction is a phase for analyzing C programs by combining four program analysis techniques. The inputs of this phase are the *Source code* and the *Formalized hardware knowledge* extracted in the first phase. The output is *Formalized software knowledge*. For formally describing the software knowledge, we propose *Specification language of software knowledge*. Detail of the specification language can be found in Section 4.3. Automatically analyzing the source code and extracting the software knowledge requires a certain level of flexibility and sophistication. Flexibility is necessary because the knowledge to be extracted depends on the target coding rules, which have many possible variations. Additionally, the solution should easily adapt to new coding rules or hardware. Sophistication is required because embedded systems have grown in size and complexity. Additionally, extracting several types of knowledge requires

deep analyses of the systems. As it is difficult for a single technique to be flexible and sophisticated, integrating multiple techniques with different strengths and weaknesses is a promising approach. This work proposes an algorithm combining four well-established program analysis techniques to extract the behaviors of source code. These program analysis techniques include pattern matching (PM), abstract interpretation-based static program analysis (AI), bounded model checking (BMC), and counterexample-guided abstract refinement (CEGAR). First, PM is employed in the form of code patterns to detect potentially related expressions of coding rules in the target source code. Secondly, AI, BMC, and CEGAR are employed to examine whether the potentially related expressions relate to a violate the coding rules. The approach to analyzing the C program can be found in Chapter 4.

Knowledge Modeling is a phase for modeling hardware knowledge and results of program analysis tools. The input of this phase is the *Formalized hardware knowledge* extracted in the first phase, the *Formalized software knowledge* extracted in the second phase, *Formalized compiler knowledge* of the used compiler, and a *Meta-model* which defines a structure to model the input knowledge. The specification language of the *Formalized compiler knowledge* can be found in Definition 5.1.1. The *Meta-model* can be found in Section 5.2. The output is a *Knowledge model* which models the three kinds of knowledge. Embedded systems are the integration of hardware and software components. Verifying these systems usually requires the combination of multiple sources of information, i.e., source code analysis results, hardware, and compiler knowledge. Additionally, multiple program analysis techniques/ tools for obtaining software information have different strengths and weaknesses. However, handling the analysis results of these techniques/tools is difficult as their outputs are usually different in both format and content. We propose a meta-model that systematically gathers the knowledge of hardware, compiler, and analysis results of program analysis tools. The meta-model is designed to adapt to different output formats of multiple knowledge extraction techniques/ tools. Ecore format, which is equivalent to a subset of Unified Modeling Language (UML) notations, is employed for designing and representing the meta-model. There are multiple advantages to applying the knowledge models for verifying hardware-dependent properties. The first advantage is that the knowledge models can be reused for other purposes, such as system understanding. The second advantage is that the meta-model is flexible in handling outputs of other program analysis tools/ techniques and easy to extend for other types of hardware-dependent knowledge. The details of this phase can be found in Chapter 5.

Lastly, *Knowledge Querying* is a verification process that utilizes the extracted knowledge and the meta-model for verifying the target system against the coding rules. The input in this phase is a *Knowledge model* which is generated from

the previous phase, a *Coding rule* in the form of a *Query*, a list of *Pre-defined queries*. The output of this phase is the *Verification result* of the target coding rule. Specifically, we apply queries over the *Knowledge model* to check for the target coding rules. As the coding rules can be categorized into several groups, pre-defined queries for the target categories of coding rules are proposed. Details of the verification approach are shown in Chapter 6.

1.4 Contribution

This research has four main contributions. The first contribution is in analyzing C programs, in which we proposed a combination of four program analysis techniques for analyzing C programs. The second contribution is in modeling microcontroller-based systems, in which we proposed a meta-model for modeling the extracted information from a hardware manual and source code. The third contribution is in verifying the target coding rules, in which we proposed a program verification framework for verifying the embedded system against the microcontroller-specific coding rules. The fourth contribution is in implementing the approach, in which we proposed an industrial-strength tool based on the verification framework.

Our first contribution is an effective combination of four program analysis techniques for analyzing embedded source code, enabling program verification for microcontroller-specific coding rules. Automatically verifying these coding rules requires a certain level of sophistication. Sophistication is required because software systems have grown in size and complexity. Additionally, verifying many coding rules requires deep analyses of the systems. While many theories and techniques have been proposed for handling hardware-dependent systems, there is a gap in practical usage for these specific coding rules. This work proposes a new algorithm combining four well-established program analysis techniques to handle a practical problem (i.e., analyzing the microcontroller-based source code). This combination successfully handled an industrial source code in the automotive field.

Our second contribution is a meta-model for modeling and verifying the compliance of microcontroller-specific coding rules. There are several works in representing information of programs, such as a fine-grained database for representing general information of source code. However, to the best of our knowledge, there is no existing solution for microcontroller-specific systems. Although we target a specific problem, this is a significant contribution when considering the population and importance of microcontroller-based systems.

Our third contribution is a program verification framework for verifying the compliance of microcontroller-specific coding rules. This framework gathers the strengths of multiple program analysis tools and model-driven engineering

techniques. In general, existing works tend to create new tools or extend existing tools. This task is very time-consuming. Developing a new industrial-strength tool from scratch for properties with a huge number of variations, like microcontroller-specific coding rules, is not practical. Our program verification framework is more lightweight and flexible than previous works.

Our fourth contribution is an industrial-strength tool for verifying embedded systems against microcontroller-specific coding rules. Innovation in science is discovering new knowledge and how we can collaborate and implement this knowledge in practice to improve the quality of human life. By contributing a tool to solve an urgent problem in practice, we have brought advances in software verification to solve practical problems. Software verification theory is highly established through a long history of development. However, there is a big gap between theory and practice. An industrial setting is much more complex in comparison with laboratory environments. Many methods proposed in academics could work well with small benchmark source codes, but not many are applicable to industrial applications. Despite that, we succeeded in applying the software verification to the source codes developed and used in a company. Our work can be considered one of the efforts to push software verification further in its evolution.

1.5 Dissertation organization

In chapter 1 (this chapter), we first introduce the research background and the research problems in focusing on microcontroller-based systems and handling microcontroller-specific coding rules. Then, we explain the overview of our proposed solutions and summarize our contributions.

Chapter 2 describes the background of this research. This section first introduces important features of microcontroller-specific systems. Next, we introduce the techniques and tools used in this research.

Chapter 3 describes the approach for analyzing hardware manuals and extracting hardware knowledge. This chapter first defines the formalized format of hardware knowledge. Then the approach for automatically extracting the target hardware manual is explained.

Chapter 4 focuses on the task of extracting knowledge from microcontroller-based source code. In this chapter, we first compare the selected techniques in terms of well-suitedness and the employed approximation approach. Then, we define the formalized format of software knowledge. Next, we explain the proposed algorithm, which combines these techniques for extracting the target knowledge.

Chapter 5 presents the approach for representing knowledge of microcontroller-based systems. This chapter explains our meta-model for representing the hardware, compiler, and software knowledge in the form of a knowledge model.

Chapter 6 details how the knowledge model is used for verifying microcontroller-specific coding rules. In this chapter, we first explain the overview of verification processes using this model. Then, we explain the pre-defined queries for the target categories of coding rules.

Chapter 7 focuses on an verification tool developed based on the proposed solution. In this chapter, we explain the implementation architecture. External tools and programming languages used in this implementation are described as well.

Chapter 8 is for evaluating the proposed approach. This chapter shows the experiments result of applying the approach to handle a benchmark source code and an industrial source code. Then we discuss several aspects of the approach.

Chapter 9 discusses the related works. We discuss the related research in three main issues: analyzing hardware manuals, analyzing and modeling source code knowledge, and verifying microcontroller-based systems. We also compare our verification tool with several existing tools for C programs.

Chapter 10 consists of conclusions and future directions. We first conclude the results of the overall dissertation. In the end, we discuss open problems and draw work that should be considered in the future.

Chapter 2

Preliminaries

This chapter explains concepts and techniques which are primarily needed to understand the proposed verification framework. As this research targets a type of coding rules for embedded systems written in C programming language, Section 2.1 first explores other well-known coding standards for these systems. Secondly, Section 2.2 introduces our target coding rules and compares our target with the well-known coding standards. Thirdly, Section 2.3 discusses the current verification process for the target coding rules. Fourthly, Section 2.4 explains how to perform an essential task in developing microcontroller-based systems, which is accessing registers. Fifthly, Section 2.5 and Section 2.6 introduce the employed techniques in the verification framework, including program analysis and model-driven engineering techniques. Finally, Section 2.7 introduces other notations used, including Backus-Naur from and linear temporal logic.

2.1 Coding standards for embedded systems

As C programming language is frequently used for safety-critical systems, several coding standards should be followed when designing and developing these systems. These standards are designed to improve code readability, ensure the code is efficient, and facilitate the development and debugging process. In this section, we discuss two common coding standards for embedded systems: the MISRA C 2012 coding standard [9] and the CERT C [10].

The MISRA C 2012 [9] is a set of coding guidelines developed by the Motor Industry Software Reliability Association (MISRA) consortium to provide recommendations for improving the safety and reliability of software developed in the C programming language. Initially, MISRA C targeted automotive systems. However, the coding standard has evolved to a wide range of embedded systems. Adhering to this standard can help ensure the quality and reliability of code written

for embedded systems. The MISRA C 2012 is the latest version of this guideline. This edition contains 143 rules and 16 directives, which were classified as *Mandatory*, *Required*, or *Advisory*. These rules and directives were also categorized based on the rules and directives' contents, such as declarations and definitions, naming conventions and commenting, or pointers and arrays. Among the standard categories, the categories related to using pointers are close to our target coding rules.

While MISRA C focuses on the safety of embedded systems, CERT C Secure Coding Standard [10] is a standard on the security aspect of these systems. The CERT C standard follows a community-based development process managed by the Software Engineering Institute (SEI). The CERT C guidelines can be found on the CERT Secure Coding wiki [10]. CERT C guidelines are classified as *Rules* and *Recommendations* [11]. A guideline is considered a *Rule* if the following three conditions are met. The first condition is that a violation of this guideline will likely cause a defect. The second condition is that conformance can be established without requiring additional assumptions. The third condition is that the guideline can be checked using automated analysis or manual inspection techniques. A guideline is considered a *Recommendation* if the conformance of this guideline likely improves the safety, reliability, or security of software systems, and additional requirements may be needed to confirm a violation of this guideline.

These coding standards can be checked manually in the code review phase or automatically using static program analysis tools. Many program analysis tools support checking the standards. We sample three popular static program analysis tools which implement different techniques. These tools are Flawfinder [12] - a lexical program analysis tool, PolySpace Bug Finder [13] - a data flow analysis tool, and CodeQL [14] - a code query tool.

Flawfinder [12] is a lexical program analysis tool to identify potential security flaws. This tool builds and uses a database of C functions with well-known problems, such as buffer overflow risks, race conditions, and poor random number acquisition. The database is built using a syntactic feature of the source code only; sophisticated analyses such as data flow analysis are not performed. This tool does not directly support MISRA c 2012 and CERT C. However, syntactic-based and several semantic-based coding rules can be checked by issuing queries over the built database.

PolySpace Bug Finder [13] is a sophisticated static program analysis tool that detects and prevents software defects and security risks in code. This tool supports checking coding rule standards such as MISRA C 2012 and CERT C.

CodeQL [14] combines a sophisticated query language with multiple analyses to identify potential flaws and unexpected behaviors. This tool first analyzes the source code and creates a database, including a representation of the abstract syntax tree, the data flow graph, and the control flow graph. Then, queries can be

issued to check for target coding rules. Queries are supporting MISRA c 2012 and CERT C.

One of the problems of static program analysis tools is that the tools often generate many false warnings [15]. Lexical program analysis tools like Flawfinder may report many false positives due to the simplicity of the technique [16]. Hence these tools are usually used for a quick look at the source code only. Data-flow analysis tools like PolySpace Bug Finder is more sophisticated. However, these tools usually try to reduce the number of false positives by over-approximating the data and control flow. This approach may lead to the analysis of some infeasible paths. CodeQL also provides a sophisticated analysis of source code. Additionally, query techniques are provided to enable writing custom checks. Custom queries written by users may help to reduce the number of false negatives, as the users usually have a good understanding of the source code. However, there is still a trade-off between false positives and false negatives. Generic queries may detect more actual bugs but also generate many false positives [17].

2.2 Microcontroller-specific coding rules

A microcontroller is a chip that integrates a processor(s), memory, and several programmable input/output peripherals. Building microcontroller-based applications is building a system at a low level. Specifically, developers must directly access the microcontroller's hardware components, such as registers or I/0 services. This process requires an understanding of hardware-dependent features. Microcontrollers are usually supplied with a hardware manual that describes these features carefully. Hardware manuals are documents to understand the microcontrollers' correct usage clearly. The hardware manual is written in natural language and often contains thousands of pages. The content of the hardware manual is mainly the explanation of the functions of the microcontrollers. Through the explanation for each function, notes and coding rules are emphasized to require special attention in using the function. Besides the coding standards such as MISRA C or CERT C, microcontroller-based systems must follow these notes and coding rules. This section gives an introduction to the microcontroller-specific coding rule using several microcontroller-specific examples. A deeper analysis of the coding rules can be found later in Chapter 3.

2.2.1 Examples of microcontroller-specific coding rules

Figure 2.1 shows several coding rules in the investigated hardware manual. They are typical examples of coding rules obtained from the hardware manual of the target register. These examples are actual coding rules; however, the name of

1. Bits at indexes from 0 to 1 of register1 are reserved bits. When reading a reserved bit of registers, an undefined value is returned.

2. Bits at index 15 of register2 is an unused bit. The value after reset of this bit is 0. When reading an unused bit, the value after reset is returned. When writing to an unused bit of registers, write the value after reset.

3. When register3[4] is set to 1, register4[21:20] and register4[5:4] should be set to 11_B and 00_B, respectively.

4. After selecting the alternative function by setting the register5[n] to 1, register6[n] to 1, and register7[n] to 1, set the register8[n] bit to "1".

5. When using register9[n] as an alternative output function (register10[n] = 1, register11[n] = 0), the level of the register9[n] pin can be read at the register12[n] bit by enabling bidirectional mode (register13[n] = 1).

6. When the RESETOUT function is selected for the register14[0] pin, register14[0] pin outputs a low-level while a reset is asserted and continues to output low level after the reset is released.

Figure 2.1: Example of coding rules

```
1  #define REG_1_ADD     0xFFF10300
2  #define INIT_VALUE        0x00000000
3
4  void func_write_32 (unsigned long add, unsigned long
       data){
5      (*(( unsigned long*)add)) = (unsigned long)data;
6  }
7
8  void main(void) {
9      unsigned long * register_1 = (unsigned long*) 0
           xFFF10300;
10     func_write_32(REG_1_ADD, INIT_VALUE);
11     unsigned long shiftExpression = 1;
12     unsigned long maskExpression = 1;
13     if (! ((*register_1 >> shiftExpression) &
           maskExpression))
14         *register_1 |= 0x00000010;
15 }
```

Figure 2.2: Example of code violating the first coding rule in Figure 2.1

the registers are changed, and the functionalities of the registers are hidden. We will explain these coding rules one by one. With each coding rule, an example of source code that violates the coding rule is provided (if any) to facilitate the understanding of the coding rule.

The first coding rule is the requirement related to reserved bits of a register. In the microcontroller, reserved bits are bits that are not used for any function in the current version of this microcontroller. These bits are used for future processors and the functionality of them have a future effect that is unpredictable at this moment. When a reserved bit is read, the undefined value or the value after the reset of this bit is returned. Performing this task is dangerous since it can lead to unexpected behaviors of the application. In the coding rule, the bit at index 0 and 1 of the register which is named register1 are reserved bits. If these bits are read, an undefined value is returned. Hence, software that is written on top of the microcontroller should not perform read-access on these bits. Figure 2.2 shows an example of a non-compliant code. In which, line 13 tries to read the value of the bit at index 1 of register1. Hence, this line violates the coding rule.

The second coding rule is the requirement related to write-access. In the microcontroller, the unused bit is the other name of the reserved bit. The bit at index 15 of register2 is an unused bit. However, if we read from this bit, the value

```
1  #define REGISTER_2_ADD     0xFFF10500
2  #define INIT_VALUE         0x0000
3
4  void func_write_16 (unsigned short add, unsigned short
       data){
5      (*(( unsigned short*)add)) = (unsigned short)data;
6  }
7
8  void main(void) {
9      func_write_16(REGISTER_2_ADD, INIT_VALUE);
10     unsigned short * register_2 = (unsigned short*)
           REGISTER_2_ADD;
11     *register_2 = 0xFFFE;
12 }
```

Figure 2.3: Example of code violating the second coding rule in Figure 2.1

after reset of this bit is returned. If we write to this bit, the valid written value is the value after reset. In this case, the value after reset of this bit is zero. Figure 2.3 shows an example of a non-compliant code. In this example, line 11 tries to change the value of register2. In which, the value of the bit at index 15 is set to 1. Hence, this line is a violation of the coding rule.

The third coding rule is a coding rule related to writing access too. However, there is a requirement for two registers instead of one register. The two registers in this coding rule are register3 and register4. This coding rule requires that when the bit at index 4 of register3 is set to 1, we need to set the bit at index 21, 20, 5, and 4 to 1, 1, 0, and 0 respectively. The order of accessing these registers is not restricted. Figure 2.4 shows an example of a non-compliant code. In this example, after the bit at index 4 of register3 is set to 1 at line 12, the value of register4 is changed at line 6. In which the values of bits at index 4 and 5 are set to 1. As this is the invalid value for this bit, this line violated the coding rule. In the case that the bit at index 4 of register3 is set to 1; however, there is no statement in the source code that set the value of the four bits of register3 to their expected value, the source code also violates the coding rule.

The fourth coding rule is the third coding rule related to performing write-access in the list of examples of coding rules. In this coding rule, there is a constraint on accessing four registers. The four registers are register5, register6, register7, and register8. In which, the requirement on the written value of register8 depends on the value written to register5, register6, and register7. Specifically, when the bits at index n of register5, register6, and register7 are all set to 1, we

```
1  #define REGISTER_3_ADD     0xFFF20338
2  #define REGISTER_4_ADD     0xFFF20304
3  #define INIT_VALUE         0x00000000
4
5  void func_write_32 (unsigned long add, unsigned long
       data){
6       (*((unsigned long*)add)) = (unsigned long)data;
7  }
8
9  void main(void) {
10   *(unsigned long*) 0xFFF20338 = (unsigned long)
         INIT_VALUE;
11       unsigned long * register3 = (unsigned long*)
             REGISTER_3_ADD;
12       *register3 = 0x00000010;
13       func_write_32(REGISTER_4_ADD, 0xEFFFFFFF);
14 }
```

Figure 2.4: Example of code violating the third coding rule in Figure 2.1

need to set the bit at the same index of register8 to 1. In this example, the order of accessing these registers is restricted. That is, register8 must be accessed before register5, register6, and register7. Figure 2.5 shows an example of a non-compliant code. In this example, after the values of the bits at index 0 of register5, register6, and register7 are set to 1 at lines 12, 13, and 14, the value of bit 0 of register8 is set to 0 by calling "func_write_32" at line 16. As this is the invalid value for register8, this line violated the coding rule. Similar to the coding rule number 3 in Figure 2.1, for this coding rule, if a source code contains statements that set the bit at index 1 of register5, register6, and register7 to 1; however, no statements that set the bit at index 1 of register8 to 1, this is also a non-compliant source code.

The fifth coding rule is a guideline to use register9, register11, register12, and register13. Specifically, register12 has several roles. One of the roles is that register12[n] is used to get the level of register9[n] when register9[n] is used as an alternative output function (i.e., when register10[n] is set to 1 and register11[n] is set to 0). To use register12 with this purpose, one has to enable bidirectional mode (i.e., set the corresponding bit in register13 to 1). To verify this coding rule, it is necessary to know the developers' intentions regarding the role used for register12. The source code in Figure 2.6 shows a source code that violates the coding rule in the case that the register12[0] is used to get the level of register9[0] when this register is used as an alternative function. In this example, after register10[0] is

```
1  #define REGISTER_5_ADD    0xFFF20100
2  #define REGISTER_6_ADD    0xFFF20200
3  #define REGISTER_7_ADD    0xFFF20300
4  #define REGISTER_8_ADD    0xFFF20400
5  #define INIT_VALUE        0x00000000
6
7  void func_write_32 (unsigned long add, unsigned long
       data){
8      (*(( unsigned long*)add)) = (unsigned long)data;
9  }
10
11 void main(void) {
12     *(unsigned long*) REGISTER_5_ADD = (unsigned long)
           0x00000001;
13     *(unsigned long*) REGISTER_6_ADD = (unsigned long)
           0x00000001;
14     *(unsigned long*) REGISTER_7_ADD = (unsigned long)
           0x00000001;
15
16     func_write_32(REGISTER_8_ADD, INIT_VALUE);
17 }
```

Figure 2.5: Example of code violating the fourth coding rule in Figure 2.1

```
1  #define REGISTER_9_ADD     0xFFF20500
2  #define REGISTER_10_ADD    0xFFF20600
3  #define REGISTER_11_ADD    0xFFF20700
4  #define REGISTER_12_ADD    0xFFF20800
5  #define REGISTER_13_ADD    0xFFF20900
6  #define INIT_VALUE         0x00000000
7
8  void func_write_32 (unsigned long add, unsigned long
       data){
9      (*((volatile unsigned long*)add)) = (unsigned long
           )data;
10 }
11
12 int main(void) {
13     func_write_32(REGISTER_10_ADD, 0x00000001);
14     func_write_32(REGISTER_11_ADD, 0x00000000);
15     int level = register12[0] & 1;
16     return level;
17 }
```

Figure 2.6: Example of code may violating the fifth coding rule in Figure 2.1

set to 1 at line 13, and register11[0] is set to 0 at line 14, register9[0] is used as an alternative output function. However, line 15 tries to read the value of register12[0] without setting register13[0] to 1. This will be a violation if the intention of line 15 is to get the level of register9[0].

The sixth coding rule is a note on the behavior of register14[0] when the RESETOUT function is selected for register14[0]. Specifically, when register14[0] is used as the RESETOUT function, register14[0] outputs a low-level while reset is asserted and released. No property needs to be examined in this coding rule.

2.2.2 Comparing with coding standards

The difference between the coding standards in Section 2.1 and our target coding rules is that the coding standards focus on the features of the used programming language while our target focuses on features of the used hardware.

In our target coding rules, we focus on special expressions of pointer-access (i.e., register-access) and the values in these expressions. The coding standards have coding rules for the safe usage of pointers in C language. MISRA C 2012 and CERT C have rules for using pointers (e.g., the conversion of pointers and integers, MISRA C 2012 Rule 11.4 and CERT C INT36-C) or value analysis (e.g., divide by zero, MISRA C 2012 Dir 4.1 and CERT C INT33-C).

Rule 11.4 and INT36-C require that a pointer not be converted into an integer and vice visa as these conversions may result in undefined behavior. Figure 2.7 shows an example source code that violates Rule 11.4 and INT36-C. At the right-hand side of line 2, a `uint32_t` typed number is converted to a `uint16_t` typed pointer. There is a case where the number and the pointer are not correctly aligned. At line 3, a pointer `p` is converted to a `uint16_t` number. The size of a pointer can be greater than the size of `uint16_t` as the size of a pointer can be 64 bits in some implementations. Hence, this line violates the coding rules in these implementations.

A part of Dir 4.1 and INT33-C require that *"Ensure that division and remainder operations do not result in divide-by-zero errors"*. Line 7 in Figure 2.7 violates this coding rule as the value of `n1` at this line is `0`.

If this source code in Figure 2.7 is implemented on hardware with a coding rule *"0x0000 should not be written to REG"*, line 4 violates this rule. As we can see, the coding standards define a safe way of using pointers in C language. Differently, our target coding rules are how to implement hardware features using C language correctly.

The similar difficulty in handling the coding standards and our target is that we have to handle complex operators in C programs such as loops or pointer analysis. However, handling our rules is easier regarding the set of expressions to be examined as we only need to focus on the limited set of expressions and

```
1 uint32_t REG = 0xFFFFFFFF;
2 uint16_t *p = ( uint16_t * ) REG; // Noncompliant
3 uint16_t number = (uint16_t) p; // Noncompliant
4 *p = 0x0000;
5 uint16_t n2 = 0x0001;
6 uint16_t n1 = *p;
7 uint16_t n3 = n2/n1;
```

Figure 2.7: Example code of violating Rule 11.4 and INT36-C

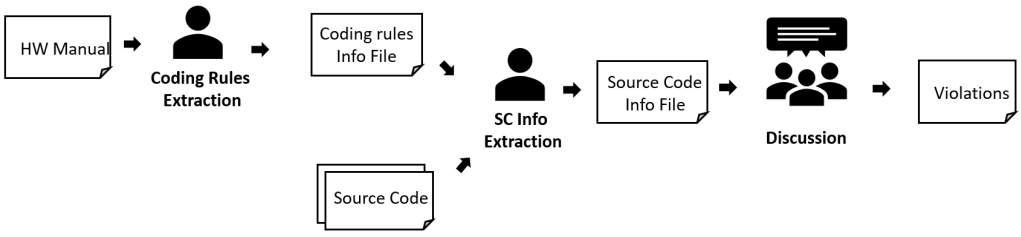

Figure 2.8: Manual verification process

these expressions have some common formats. However, the more difficult task is that we need to check these expressions with a large variance of properties which depends on the used hardware. It means that our solution must be able to adapt to different hardware models.

On the other hand, our target is how to correctly perform register-access, which is related to pointer-access, on a hardware model. Specifically, we focus on special forms of pointer-access operators and how these operators are correctly used for performing register-access regarding the hardware used. The main difficulty in automatically handling the two coding standards is the complexity of C language. We also need to handle this difficulty for our target. However, adding to this difficulty, we need to handle various hardware models too.

2.3 Current verification process for microcontroller-specific coding rules

Currently, verifying those coding rules is an entirely manual process. Figure 2.8 shows this manual verification process for the coding rules. There are three phases: **Coding Rules Extraction**, **SC Info Extraction**, and **Discussion**. In the **Coding Rules Extraction** phase, a developer first extracts a coding rule with its

related parts in the *HW Manual* (i.e., hardware manual). The coding rules are often marked with the keywords "note" or "caution" and usually relate to other parts of the manual. The result of this phase is *Coding Rules Info File* - a file that contains the target coding rule and the related parts. For instance, as the example coding rules above refers to `register_1` and `register_2`, the *Coding Rules Info File* in this case contains the coding rule and the information of these two registers. In the **SC Info Extraction** phase, a developer manually examines every statement in the source code and marks all statements related to the target coding rule. We take the source code in Figure 1.1 and the example coding rule above to explain this process. Specifically, the developer analyzes the source code, finds lines that access related registers of the coding rules (e.g., lines `7` and `9`), and stores the source code annotated with the information about register-access in the *Source Code Info File*. In the **Discussion** phase, a group of developers gathers and discusses whether the source code violates the coding rule based on the *source code info file*. The manual verification process is time-consuming and costly in terms of human resources, as the participants must have a solid knowledge of both the source code and the microcontroller. This manual process makes the verification phase the most time-consuming phase in the development process.

2.4 Manipulating registers of micorocontrollers

An important task in implementing a microcontroller-based system is to manipulate the registers. There are several methods for manipulating the registers of microcontrollers [18]. The first method is memory-mapped I/O in which the device registers are mapped to conventional data space. The second method is port-mapped I/O, in which control and data registers are mapped to separate small data spaces. Additionally, several microcontroller families provide special methods of accessing and manipulating the memory via the I/O mapped.

Among the methods for manipulating the registers, the memory-mapped I/O is usually performed using C or C++ programming language. The two other methods usually require non-standard language or special library features. We limit our research to microcontroller-based systems written in C programming language, so we only explain the memory manipulation method using this language.

Microcontroller-based systems are usually written in C and assembly. The majority are usually implemented in C as writing in assemble tediously and error-prone [19]. In C programming language, registers are represented as normal data. The language has several features which support implementing embedded systems, such as pointers for declaring the addresses and values of registers or `bitwise` operators for manipulating specific bits in registers. For example, in Figure 2.9, the register at address `0xFF70` is represented by a variable named `pcr`. The size

```
1 typedef unsigned short control;
2 #define ENABLE 0x0080 /* bit 7: ready mode */
3
4 control *const pcr = (control *)0xFF70;
5
6 *pcr &= ~ENABLE;
```

Figure 2.9: Example code of manipulating registers

of this register is represented by the data type `unsigned short`. Assume that the size of the register at `0xFF70` is 16 bits and `unsigned short` is 16 bits datatype in the compiler of this system, `unsigned short` will be the representation for the size of this register. The register is manipulated in line `6` to clear the ready bit in this example.

Microcontroller-based systems can be large. However, in most cases, only some fragments of code are related to a specific coding rule. One idea is to narrow down the focus to a small set of code fragments for each coding rule. In this research, we need to find program fragments that are related to register access. The difficulty in finding these fragments is variants of C source code on reading/writing registers. In an embedded system, register access operators are usually performed by directly accessing the addresses of hardware registers. The C programming language provides several features to support manipulating hardware such as pointer operator, volatile qualifier, `const` qualifier, and `bitwise` operator. In order to perform read/write register operators, we need to consider how to use C language to represent hardware registers first. In [18], Dan Saks explains methods for representing hardware registers involve individual bit manipulation. The first method is using macros to represent the addresses of hardware registers. The source code below is an example of using macro symbols to represent the hardware register. In this example, ADDRESS represents a register in which its address is (0x1200000 + 0x00) and its size is 8 bits.

```
1 #define BASE 0x1200000
2 #define ADDRESS (volatile unsigned char *) (BASE + 0
      x0000)
```

Registers can be represented as variables in C source code as in the following example.

```
1 #define BASE 0x1200000
2 #define REGISTER *((volatile unsigned char *) (BASE +
    0x0000))
3 #define MASK 0x0001
4 if ((REGISTER >> 6) & MASK) {
5 // do something if bit 6th is 1
6 }
```

(a) Example 1

```
1 unsigned char * register = (volatile unsigned char *)
    (0x1200000 + 0x0000);
2 if ((*register >> 6) & MASK) {
3 // do something if bit 6th is 1
4 }
```

(b) Example 2

Figure 2.10: Example of read-access operators

```
1 *(unsigned char*) 0x1200000 = 0x00;
```

(a) Example 1

```
1 unsigned char* register = (unsigned char*) 0x1200000;
2 *register = 0x00;
```

(b) Example 2

Figure 2.11: Example of write-access operators

```
1 unsigned long BASE = 0x1200000;
2 unsigned char * ADDRESS = ((volatile unsigned char *)
    (BASE + 0x0000));
```

For manipulating individual bits, bitmask operators can be used, as in the following example. In this example, the bitmask BIT6 is used to check the value of the bit number 6 in the register ADDRESS.

```
1 #define BASE 0x1200000
2 #define ADDRESS (volatile unsigned char *) (BASE + 0
    x0000)
3 #define BIT6 0x0040
4 if (ADDRESS & BIT6) {
5 // do something if bit 6th is 1
6 }
```

Several devices prefer grouping semantically related registers such as a control and a data register, structure in C language can be used like the following example.

```
1 #define BASE 0x1200000
2 struct port {
3   volatile unsigned char CONTROL;
4   volatile unsigned char DATA;
5 }
6 #define PORT (port *) BASE
7 PORT->CONTROL = 0x00;
```

As member CONTROL is a register at offset zero within the port struct, the expression "PORT− >CONTROL" is equivalence with an expression "(*(volatile unsigned char *) BASE)". By using structure in C language, we can also represent hardware as bitfields as in the following example.

```
1 struct reg {
2  unsigned char enable: 1;
3  unsigned char unused: 7;
4 }
5 reg volatile *reg1 = (reg*) 0x1200000;
6 reg1.enable = 0;
```

In the example above, the most significant bit (assume that the first bit is the most significant bit in the target platform) of the register at address 0x1200000 can be accessed as in line 6.

After representing registers in C language, we can access registers via their representations. In this research, we device register access operators into two groups: read-access and write-access. For read-access, the popular way to perform read-access operators is using shift operators and using a bitmask. Figure 2.10 shows examples of using the shift operator and bitmask for reading the bit at index 6 of a register.

For write-access, most embedded systems use memory-mapped I/O, which maps registers to fixed addresses in the conventional memory space so that we can modify the values of specific locations [18]. Figure 2.11 shows examples of writing a value to an address using memory-mapped I/O. In these examples, the value 0x00 is written to a register in which the address of this register is 0x1200000.

In addition to using C language to perform register-access, some particular devices provide special instructions. This method requires using a non-standard language or specific libraries. As it is outside the C language standard, we do not consider this method in this research.

2.5 Program analysis techniques and tools

Program analysis is a field of computer science that focuses on understanding and analyzing the behavior of computer programs. It is used to detect errors, optimize performance, and verify program correctness. Program analysis techniques use static and dynamic methods to analyze a program and its behavior. Static analysis techniques are used to analyze a program without actually running it, while dynamic analysis techniques analyze a program while it is running. After a long development history, various static program analysis techniques were introduced; many were successfully employed in practice. We select four highly established techniques in the field to extract knowledge of microcontroller-based systems. They are pattern matching, abstract interpretation-based static program analysis, bounded model checking, and counterexample-guided abstract refinement. Many verification tools were employed based on these techniques. Among these tools, we select a popular tool with industrial strength for each technique. A brief introduction to the selected techniques and tools is provided in this section.

2.5.1 Pattern matching (PM) and Cobra

Pattern matching (PM) is a technique used in computer science to determine if a given input matches a predefined pattern. It is often used in parsers, interpreters,

and databases. The main idea behind pattern matching is to provide a method for testing whether a given input, for example, a string, matches a given pattern. The pattern is usually defined as a set of rules that describe how the input should be structured. Pattern-matching algorithms can then determine if a given input follows the pattern. The patterns can have the form of regular expression or context-free grammar.

Regular expression matching is used to find strings that match a given pattern in the form of regular expressions [20]. This technique represents a string as a list of characters and scans over this list to detect matching sequences of characters. Currently, the regular expression function for this purpose is widely implemented in programming languages such as C, Java, and Python. In this research, regular expression-based pattern matching is employed for analyzing hardware manuals and extracting the knowledge of the target hardware.

On the other hand, pattern matching using context-free grammar is a technique where the patterns are described in context-free grammar [21]. In this research, this technique is used to analyze source code and extract fragments of code that relate to register-access. In program analysis, pattern matching is a lightweight technique. This technique is fast and flexible. However, it is imprecise as related fragments may be missed, and unrelated fragments may be detected. As examining all expressions in the source code is unnecessary and impractical, the extraction focus is narrowed down to expressions potentially related to the target knowledge. We consider PM suitable for this task.

Cobra [22, 23] is a structural source code analyzer introduced in 2015. This tool analyzes programs by applying the pattern-matching technique over lexical tokens. Lexical tokens are C language tokens annotated with other information like token types. In general, the principle of the tool is to build a data structure (that is, the linked lists of lexical tokens) to represent a source code and provide an interface for querying the source code to check target properties. The tool can quickly deal with a large source code because of its simplicity [23]. In this research, Cobra can handle the task of detecting expressions that may relate to register-access.

2.5.2 Abstract interpretation-based static program analysis (AI) and Eva plugin of Frama-C

Abstract interpretation is a theory that approximates the behaviors of programs and interprets this program based on this approximated behavior instead of the concrete one [24]. Abstract interpretation-based static program analysis (AI) can facilitate the value abstraction of the program's expressions. As for the target of analyzing expressions in the program and extracting knowledge related to register-

access, AI can calculate possible values for expressions that may be related to our target. This approach is sound but may generate a large number of false warnings as over-approximation is employed [25].

Eva plugin [26] is an abstract interpreter which combines multiple abstractions in the abstract interpretation theory to provide value and state abstractions. The value abstractions are suitable for our target of calculating possible values of expressions in programs to extract knowledge related to register-access. The state abstractions are abstractions of memory states where both low-level concepts of memory states (e.g., bit fields) and high-level ones (e.g., array) are presented [27]. These abstractions are also suitable for our target of analyzing hardware-dependent source code where bitfields and arrays are frequently used.

2.5.3 Bounded model checking (BMC) and CBMC

Model checking is a method to verify a property against a finite state machine and can be used to verify a program. Bounded model checking (BMC) is a technique that limits the state space to be searched and looks for a counterexample of a target property [28]. This method provides an under-approximation of the analyzed program. There are no false warnings generated. However, the safety is not ensured outside the searched space. This method can be used to reconfirm the results of analysis techniques that employ over-approximation. Additionally, BMC provides explanations for detected warnings in the form of counterexamples.

CBMC [29] is a bounded model checker for low-level C programs. The tool is well-suited for handling microcontroller-based systems. Additionally, it can be used as a complement with Eva plugin as these two tools implement different approximation approaches.

2.5.4 Counterexample-guided abstraction refinement (CEGAR) and SatAbs

Counterexample-guided abstraction refinement (CEGAR) [30] is an automatic iterative abstraction-refinement technique where the abstract model is an over-approximation of the concrete behavior of a program. This model may generate erroneous counterexamples. Symbolic execution is employed to eliminate these erroneous counterexamples and refine the abstract model. CEGAR is sound and does not report false warnings. However, this technique is heavy because of the iterative abstraction-refinement steps. Similar to BMC, CEGAR is suitable for providing explanations. BMC and CEGAR are expected to be the complement in handling this task.

SatAbs [31, 32] is a model checking tool which employs CEGAR. Specifically,

this tool implements a predicate abstraction and a refinement process to cope with erroneous counterexamples. Similar to CBMC, SatAbs can be used to explain the extracted knowledge.

2.5.5 Under-approximation and over-approximation

Program analysis is often executed in the abstractions of programs as it is tough to examine the concrete semantics of programs. Approximation techniques are introduced to handle this case. Approximation techniques can be divided into the under-approximation approach and the over-approximation approach.

The under-approximation approach contains techniques that under-approximate the state space of programs. Applying these techniques can analyze an important but incomplete subset of the state space. The main advantage of this approach is that the cost of analyzing programs can be reduced. However, the disadvantage of this approach is that the possibility of missing bugs is high as it is not an exhaustive approach.

By contrast, over-approximation contains techniques that over-approximate the state space of a program. The state space generated by this approach is often broader than the actual state space (i.e., includes unreachable states). The advantage of this technique is that the coverage in detecting bugs may increase. However, the disadvantage is that false warnings may be reported.

2.6 Model-driven engineering techniques and tools

In this research, we apply several model-driven engineering (MDE) techniques for representing the knowledge of microcontroller-based systems and verifying these systems against the target coding rules. Data models are centric for MDE. In MDE, a model is usually described by domain experts first. Subsequently, model transformation techniques will transform the model to a different format based on the target application. The MDE techniques employed in this research include the Unified Modeling Language (UML) as the language for describing the model, QVT operational mappings for transforming the model, and the Eclip modeling framework as the toolsets for implementing the process of describing and transforming the model.

2.6.1 Unified Modeling Language

The Unified Modeling Language (UML) is a general-purpose, developmental modeling language in software engineering that is intended to provide a standard way to visualize the design of a system [33, 34]. UML provides facilities for

creating diagrams and documents for a software system. As a modeling language, UML can be used to communicate the requirement of the system and how the system can be implemented. UML is often used in the requirements phase to provide developers with a clear understanding of user needs.

There are several kinds of UML diagrams [35] such as use case diagrams, class diagrams, object diagrams, activity diagrams, sequence diagrams, and deployment diagrams. The use case diagram can be used to describe the functional requirements of the system. The class diagram can explain the classes of object components and relations among these objects in the systems. The object diagram is similar to the class diagram but with concrete values for each object. This diagram represents specific examples for the instance of classes in the class diagram. The sequence diagram presents the message passing among objects in the systems. The deployment diagram is used to describe the software components as well as hardware devices that install these components in the actual environment. In this research, we employ the class diagram to represent the structure knowledge of microcontroller-based systems.

2.6.2 Eclipse Modeling Framework

Eclipse Modeling Framework (EMF) [36, 37] is a modeling framework for building applications based on a structured data model. EMF can be used as the foundation for interoperating among different toolsets and model-based applications. This framework consists of a set of Eclipse plugins that can be used for creating, editing, and visualizing the data model, making the model from a specification in XML Metadata Interchange (XMI), and generating Java source code from the data model. Two important terms to be distinguished in EMF are meta-model and model. A meta-model is used to describe the structure of the data model, while a model is a concrete instance of the meta-model. EMF provides several methods for creating the meta-model, including using XMI, Java assertions, UML, or an XML scheme. After creating a meta-model, instances of this meta-model can be generated using the appropriate data.

In this research, we employ EMF as the framework for modeling the information extracted from the hardware manual and analyzing source code using the four program analysis tools. We selected EMF for our task because this framework can handle a large amount of data, and the implemented UML subset is sufficient for describing the target knowledge.

2.6.3 QVT operational mapping

QVT operational mappings (QVTo) is an imperative language specified in 2007 by the Object Management Group (OMG) for model-to-model transformation

[38, 39]. In a QVTo transformation, we must first specify the source and target metamodels for the mapping. Then, the mapping among objects from instances of the two metamodels also needs to be described.

Like other general programming languages, QVTo provides imperative construction such as `for` and `while` loops, variables with scope, switch, break and return statements. Additionally, there are two popular used utilities in QVTo named query and helper. A query is a function to obtain data from objects of the input metamodel. For example, a query can be used as a filter to obtain the objects with a constraint. An important characteristic of queries is that they do not have side effects on their parameters. On the other hand, helpers are similar to queries but may have side effects.

2.7 Other notations

2.7.1 Backus–Naur Form

Backus–Naur Form (BNF) is a meta-language to define the syntax for a programming language. John Backus and Peter Naur originally introduced BNF to describe the syntax of ALGOL [40, 41].

According to the way BNF is used to define the ALGOL 60 [41], a BNF specification is a set of derivation rules. A derivation rule is in the following form:

<symbol> ::= `expression`, in which

- <symbol> is a non-terminal symbol that appears on the left-hand side and is enclosed between the pair of angle brackets.
- `expression` is a sequence of either terminal or non-terminal symbols;
- `::=` means "is defined as";
- A terminal symbol is a symbol that does not appear on the left-hand side.

Table 2.1 shows a list of notations for BNF used in this research. In this table, notations except the double quotation marks (i.e., notions numbers 0-3) are originally used in [41]. Double quotation marks are added to allow the notation numbers 0-3 to appear in terminal symbols. In this research, BNF is used to define the code patterns for loops and register-access.

2.7.2 Linear temporal logic

Linear temporal logic (LTL) [42] is a formalism for specifying a property in which time is considered linear. PLTL [43] is an extension of LTL with past operators.

Table 2.1: Notations used for Backus–Naur Form (BNF) specifications

Number	Notation	Usage
0	::=	definition
1	;	termination
2	|	alternation
3	<...>	non-terminal
4	"..."	terminal symbol

The past operator is a symbol used in LTL to refer to an event that already occurred in the past. In this research, the temporal properties are represented using a subset of PLTL operators [44] including:

- `G`: Always, `G`φ is true iff φ is true for all points in the time sequence;
- `F`: Eventually, `F`φ is true iff φ is true at some points in the future (including the present);
- `X`: Next, `X`φ is true iff φ is true at the next point;
- `O`: Once, `O`φ is true iff φ is true at some points in the past (including the present);
- `Y`: Yesterday, `Y`φ is true iff φ is true at the previous point.

This subset is used to represent the temporal properties in the target coding rules.

Chapter 3

Analyzing hardware manual

3.1 Approach overview

Microcontrollers usually come with manuals for describing the desired usage. The coding rules of these microcontrollers are usually highlighted in these manuals as notes or coding rules to call for special attention in using the hardware. As this research aims to verify the coding rules, a sufficient understanding of the hardware manual and coding rules is required.

To understand the specific characteristics of the manuals and target coding rules, we first investigate the manual of a popular microcontroller in the automotive field. This manual consists of 38 chapters and 2415 pages. The investigation was conducted in my Master's course [1]. Specifically, developers of embedded systems roughly scanned the manual and extracted the coding rules by looking for fragments with special keywords such as "note" or "caution". Subsequently, we selected two chapters that contained a sufficient number of coding rules, analyzed the coding rules, and categorized these coding rules into several groups. The result of this investigation is detailed in Section 3.2. Based on this investigation, we found that coding rules related to register-access are frequently required in the hardware manual.

In these manuals, there are sections describing the microcontroller's registers; each group of microcontrollers is in one separate section. The physical and logical information about a register is described in a register section. The physical information of the register includes the access size, the address, the value after reset, the names of bits, and the accessibility of bits. The logical information is the coding rules of using these registers.

Hardware manuals are semi-structured documents. The physical information of the registers is described in fixed formats, while the coding rules are described in natural language as the examples in Figure 2.1. We can not reveal the structure

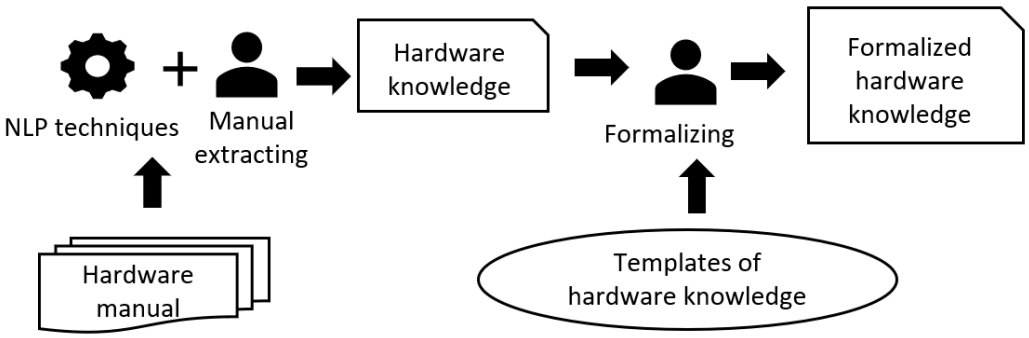

Figure 3.1: Overview of the proposed approach

of the targeted manual in this research due to the policy of AISW. However, other microcontrollers also share the regulation of describing register information in a structured format. Figure 3.2 shows examples of register sections in the manual of ATmega328, STM32F101xx, and PIC16F87XA families. We can see that although the register information is described in different formats for each microcontroller family, this information is all in structured formats. Additionally, tables are mainly used to show the information of individual bits in these examples. As our target is to verify systems against microcontroller-specific coding rules (i.e., coding rules related to register-access), we need to extract the information about registers, such as the register's addresses, register names.

There are two technical difficulties in extracting and describing the knowledge from the hardware manual. The first difficulty comes from the non-structure format of the hardware manual. The hardware manual is written in natural language, which is difficult to verify automatically. The second difficulty comes from the ambiguity of the coding rules. As coding rules of microcontrollers are written in natural language, there exists the case that one expression has several meanings.

To handle the two difficulties above, we propose a semi-automated approach that extracts and transforms the natural language requirements into formalized ones. Figure 3.1 shows the overview of the approach for analyzing the hardware manual. The input of the approach is *Hardware manual* which is a hardware manual in PDF format. The output is *Formalized hardware manual* which is the formalized knowledge related to registers. There are two steps in this approach. The first step is to analyze the hardware manual and extract knowledge on registers. The input of this step is a *Hardware manual*; the output is a *Hardware knowledge* in any format. This step involves manual effort and the automated process using natural language processing techniques. The automated process is proposed to handle the first difficulty mentioned above partly. Details of the automated process can be found in Section 3.4. The second step is to describe the extracted hardware

6.3.1 SREG – AVR Status Register

The AVR status register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

(a) ATmega328P[1]

[1]Source: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf

24.6.8 Clock control register (I2C_CCR)

Address offset: 0x1C
Reset value: 0x0000

Note: 1 F_{PCLK1} is the multiple of 10 MHz required to generate the Fast clock at 400 kHz.
2 The CCR register must be configured only when the I2C is disabled (PE = 0).

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
F/S	DUTY	Reserved		CCR[11:0]											
rw	rw			rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

(b) STM32F101xx[1]

[1]Source: https://www.keil.com/dd/docs/datashts/st/stm32f10xxx.pdf

REGISTER 2-4: PIE1 REGISTER (ADDRESS 8Ch)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PSPIE(1)	ADIE	RCIE	TXIE	SSPIE	CCP1IE	TMR2IE	TMR1IE
bit 7							bit 0

(c) PIC16F87XA[1]

[1]Source: http://ww1.microchip.com/downloads/en/devicedoc/39582c.pdf

Figure 3.2: Examples of register sections

Table 3.1: Categories of coding rules of a microcontroller in two sections [1]

Category of coding rules				Number of coding rules
Verifiable	Read-access			4
	Write-access	Non-conditional-case		24
		Conditional-case	Before	15
			After	11
			Before-or-after	13
Unverifiable	Depend on developers' intention			4
	Just a note			14

knowledge in a formalized format. The input of this step is the extracted *Hardware knowledge* in the first step and a *Specification language of hardware knowledge*. The *Specification language of hardware knowledge* is partly proposed during my Master's course [1]. This specification language is proposed to overcome the second difficulty mentioned above. The output of this step is a *Formalized hardware manual* which is the *Hardware manual* described in the *Specification language of hardware knowledge*.

In the remaining of this chapter, we first explain the investigation results of the target hardware manual. Specifically, the categories of coding rules and the specification language for formally describing the hardware knowledge will be explained in Section 3.2. Secondly, the approach for automatically analyzing the hardware manual is explained in Section 3.4.

3.2 Categories of coding rules

One of the problems in verifying the coding rules is that the number of coding rules of a microcontroller can be large (e.g., 492 fragments that describe notes or cautions in the target microcontroller). However, several coding rules share the same properties; for example, some coding rules are related to the register read-access while others are related to the register write-access. Coding rules which have the same properties may require similar approaches to handle. It is easier to handle coding rules by groups of similar coding rules than by each coding rule individually. Hence, we categorized coding rules which require a similar verification approach into the same groups.

We analyzed and categorized the coding rules extracted from two sections that contain a considerable number of coding rules in the target hardware manual. Specifically, we investigated 85 coding rules (i.e., 17% of the total number of coding rules) in these two sections. The two sections relate to the pin function and clock controller, which are two important microcontroller settings.

Table 3.1 shows the categories in the investigated part of the two sections. These categories were published in [1]. There are two groups in these 85 coding rules: *verifiable* and *unverifiable*. The *unverifiable* group contains coding rules that cannot be verified by analyzing the program only. The first reason is that, in some cases, we need to know the developers' intention to decide whether the program violates the coding rule. For this reason, the fifth example in Figure 2.1 is an example of the *unverifiable* group. The second reason is that some coding rules are just notes on the microcontroller's behavior or performance in specific cases. Hence, there is no property to be examined. For this reason, the last example in Figure 2.1 is an example of the *unverifiable* group.

Our focus is on the *verifiable* group, which can be verified by analyzing the source code. This group is divided into two sub-groups: *read-access* and *write-access*. *Read-access* are coding rules of the readability of registers and their bits. Reading a write-only or reserved bit must be avoided as it may lead to unexpected behaviors. The first coding rule in Figure 2.1 is an example that belongs to the *read-access* category, in which reading from reserved bits of `register1` is prohibited.

Write-access is the major category containing 72% (61 coding rules) of the investigated part. Based on the structure of coding rules, this category is divided into two sub-categories: *non-conditional-case* and *conditional-case*. The *non-conditional-case* coding rules are coding rules on using an individual register. The *conditional-case* coding rules are coding rules in which constraints of a register depending on the value of at least one register. There are three types of constraints on the written value of a register: on the written value of the whole register, on the written value at a bit, and on the size of the written value. The second example in Figure 2.1 is an example of a *non-conditional-case* coding rule in which the written values of several bits in `register2` are restricted as they are unused bits. The third example in Figure 2.1 is an example of a *conditional-case* coding rule where constraints on written values of several bit of `register4` depend on the written value of a bit of `register3`. The fourth example in Figure 2.1 is also an example of a *conditional-case* coding rule where constraints on the written value of a bit of `register8` depend on the written value of bits of `register5`, `register6`, and `register7` at the same index with the bit of `register8`.

In the *write-access*, one may need to consider the order of accessing registers if there are requirements for accessing more than one register (i.e., in the case of the *conditional-case* coding rules). For example, the third coding rule in Figure

2.1 does not have any requirement on the order of accessing `register3` and `register4`; however, the fourth coding rule in this figure requires that `register8` must be accessed after `register5`, `register6`, and `register7` are accessed.

In Table 3.1, we categorized the types of accessing order into three groups: *after*, *before*, and *before-or-after*. *After* means that a register must be accessed after the time that other registers are accessed; *before* means that a register must be accessed before the time that other registers are accessed; *before-or-after* means that the order is not restricted. Hence, it is necessary to provide methods to describe and verify these properties.

3.3 Specification language for hardware knowledge

As the required hardware knowledge for the verification process are the registers and coding rules on accessing these registers, we define hardware knowledge as a triple tuple of **name**, **register_info** and **coding_rules** as follows.

Definition 3.3.1 (Hardware knowledge) *A hardware knowledge structure **HWK** is a tuple **HWK** = <**name**, **register_info**, **coding_rules**>, where*

***name** ∈ **S**;*
***register_info** = [**reg_info_1**, .., **reg_info_n**], **n** ∈ ℕ∗;*
***reg_info_i** = <**regName**, **regAdd**>;*
***regName** ∈ **S**; **S** is the set of strings;*
***regAdd** ∈ **H**; **H** is the set of hexadecimal numbers;*

***coding_rules** = [**coding_rule_1**, .., **coding_rule_n**], **n** ∈ ℕ∗;*
***coding_rule_i** ∈ **SFCR**; **SFCR** is the set of formalized coding rules.*

The hardware knowledge includes registers' information and coding rules for using these registers. A name and an address represent a register. For example, the representation of a register named `register_1` with the address is `0xFFE20244` is: <*register_1, 0xFFE20244*>. The representation of hardware named microcontroller1 with `register_1` with the address is `0xFFE20244` and `register_2` with the address is `0xFFE51094` is as below.

- <*microcontroller1, [<register_1, 0xFFE20244>, <register_2, 0xFFE51094>]*>

Coding rules related to register-access are represented in a format of formalized coding rules. A formalized coding rule is a double tuple of **category** and **requirement** as follows.

Definition 3.3.2 (Coding rule) *A coding rule CR is a tuple*

CR = <***category***, ***requirement***>, *where*

category = *read-access* | *non-conditional-case* | *conditional-case;*

requirement = ***read-access-constraint*** | ***p*** | ***p*** *before* ***q*** | ***p*** *after* ***q***| ***p*** *before-or-after* ***q****;*
p = ***write-access-constraint****;*
q = ***write-access-constraint*** | ***q*** ∧ ***q****;*

read-access-constraint = ***NotReadable*** | ***NotReadableBit*** | ***ReadValidSize*** | ***ReadInvalidSize****;*
NotReadable = *Register* ***regName*** *is not readable;*
NotReadableBit = ***regName[i]*** *is not readable;* ***i***∈*[0,..,****regSize*** *- 1];*
ReadValidSize = *Read size of register* ***regName*** *belongs to [****size0****, ..,****sizen****];* ***sizei*** ∈ *{8, 16, 32, 64};*
ReadInvalidSize = *Read size of register* ***regName*** *does not belong to [****size0****, ..,****sizen****];* ***sizei*** ∈ *{8, 16, 32, 64};*

write-access-constraint = ***ValidValue*** | ***InvalidValue*** | ***WriteValidSize*** | ***WriteInvalidSize*** | ***ValidValueBit*** | ***NotWritable*** | ***NotWritableBit****;*
ValidValue = *Written value of register* ***regName*** *belongsto [****val0****, ..,****valn****];* ***vali*** ∈ ℕ*;*
InvalidValue = *Written value of register* ***regName*** *does not belong to [****val0****, ..,****valn****];* ***vali*** ∈ ℕ*;*
WriteValidSize = *Written size of register* ***regName*** *belongs to [****size0****, ..,****sizen****];* ***sizei*** ∈ *{8, 16, 32, 64};*
WriteInvalidSize = *Written size of register* ***regName*** *does not belong to [****size0****, ..,****sizen****];* ***sizei*** ∈ *{8, 16, 32, 64};*
ValidValueBit = *[****regName****[i] is* ***valBit*** | ***i***∈*[0,..,****regSize*** *- 1];* ***valBit***∈*{0,1};]*
NotWritable = *Register* ***regName*** *not writable;*
NotWritableBit = ***regName[i]*** *is not writable;* ***i***∈*[0,..,****regSize*** *- 1];*

Definition 3.3.3 (Temporal properties) *The temporal properties are defined as follow:*
p *before* ***q*** : $\boldsymbol{G}(\boldsymbol{q} \Rightarrow O\,Y\,\boldsymbol{p})$*;*
p *after* ***q*** : $\boldsymbol{G}(\boldsymbol{q} \Rightarrow \Diamond\, X\,\boldsymbol{p})$*;*
p *before-or-after* ***q****:* $\boldsymbol{G}(\boldsymbol{q} \Rightarrow (O\,Y\,\boldsymbol{p}) \vee (\boldsymbol{F}\,X\,\boldsymbol{p}))$*.*

The **category** element indicates how to verify a coding rule, as different verification algorithms are provided for different categories. There are three possible values for this element corresponding with categories of the coding rules: *read-access*, *non-conditional-case*, and *conditional-case* corresponding with categories defined in Table 3.1. The *non-conditional-case* and *conditional-case* are for *write-access*. In Figure 2.1, the value of this part for the first coding rule is *read-access*; for the second coding rule is *non-conditional-case*; for the third and fourth coding rules is *condition-case*.

The **requirement** element indicates the required properties of a coding rule. For each coding rule category, corresponding formats are provided to describe the **requirement** element. For the *read-access* category, there are three kinds of requirements: the readability of a whole register, the readability of a bit of a register, and the access sizes of a register. The requirement can be expressed as a list of valid access sizes or a list of invalid access sizes for access sizes. For example, the **requirement** part of the first coding rule in Figure 2.1 is *`register1[15]` is not readable*. The full formalized coding rule, in this case, is as follows.

- <*read-access*, *`register1[0]` is not readable*>
- <*read-access*, *`register1[1]` is not readable*>

For the *write-access*, there are five kinds of requirements: on the written values of a whole register, on the written values of a bit of a register, on the access sizes of a register, on the writability of a whole register, and the writability of a bit of a register. For the written value (or the access size of a register), the requirement can express a list of valid values or a list of invalid values (or a list of valid access sizes or a list of invalid access sizes).

For the *non-conditional-case* coding rule in the *write-access* category, the **requirement** element is used to describe the constraint on how a write-access performed on a register. The **requirement** can be on the size of the written value, the written value, or the written value of a bit. For example, the **requirement** part of the second coding rule in Figure 2.1 is *`register2[15]` is 0*. The full formalized coding rule, in this case, is as follows.

- <*non-conditional-case*, *`register2[15]` is 0*>

For *conditional-case* coding rules, there are two parts: the condition and the requirement. For example, in the third coding rule in Figure 2.1, the condition is that `register3[4]` is set to 1; the requirement is that `register4[21:20]` and `register4[5:4]` are set to 11_B and 00_B respectively. The condition and

requirement parts can be represented in the same formats as the *non-conditional-case* coding rules. However, for the *conditional-case* coding rule, one must consider the temporal properties to represent the relationship between the condition and requirement parts. The temporal properties are defined in Definition 3.3.3. These properties are defined using a subset of PLTL operators (LTL with past operators) [44] including `G`: Always, `F`: Eventually, `X`: Next, `O`: Once, `Y`: Yesterday. This subset is sufficient to represent the coding rules with their temporal properties (i.e., *before*, *after*, *before-or-after*).

As the coding rule example is split into four *conditional-case* coding rules: *When* `register3[4]` *is set to 1,* `register4[21]` *should be set to 1*, *When* `register3[4]` *is set to 1,* `register4[20]` *should be set to 1*, *When* `register3[4]` *is set to 1,* `register4[5]` *should be set to 0*, *When* `register3[4]` *is set to 1,* `register4[4]` *should be set to 1*, we take the first coding rule as the example to show how to represent the **requirement** part for this category. The **requirement** part of the first coding rule is `register4[21]` *is 1 before-or-after* `register3[4]` *is 1*. The full formalized coding rule, in this case, is as follows.

- <*conditional-case*, `register4[21]` *is 1 before-or-after* `register3[4]` *is 1*>

Similar to the third coding rule in Figure 2.1, the fourth coding rule is also a *conditional-case*, however, with three conditions. The difference is that the condition part in the fourth coding rule is the conjunction of multiple conditions. For the last coding rule in Figure 2.1, taking `n` is `0`, the **requirement** part is written as `register8[0]` *is 1 after (*`register5[0]` *is 1* $\wedge$ `register6[0]` *is 1* $\wedge$ `register7[0]` *is 1)*. The full formalized coding rule, in this case, is as follows.

- <*conditional-case*, `register8[0]` *is 1 after (*`register5[0]` *is 1* $\wedge$ `register6[0]` *is 1* $\wedge$ `register7[0]` *is 1)*>

The full formalization of the hardware with `register1` and the first coding in Figure 2.1 is as follows.

- <microcontroller, [<`register1`, `0xFFF10300`>], [<*read-access*, `register1[0]` *is not readable*>, <*read-access*, `register1[1]` *is not readable*>]>

The full formalization of the hardware with `register2` and the second coding in Figure 2.1 is as follows.

- <microcontroller, [<`register2`, `0xFFF10500`>], [<*non-conditional-case*, `register2[15]` *is 0*>]>

The full formalization of the hardware with `register3` and `register4` and the third coding in Figure 2.1 is as follows.

- <microcontroller, [<`register3`, `0xFFF20338`>, <`register4`, `0xFFF20304`>], [<*conditional-case*, *`register4[21]` is 1 before-or-after `register3[4]` is 1*>]>

The full formalization of the hardware with `register5`, `register6`, `register7`, and `register8`, and the fourth coding in Figure 2.1 is as follows.

- <microcontroller1, [<`register5`, `0xFFF20100`>, <`register6`, `0xFFF20200`>, <`register7`, `0xFFF20300`>, <`register8`, `0xFFF20400`>], [<*conditional-case*, *`register8[0]` is 1 after (`register5[0]` is 1 $\wedge$ `register6[0]` is 1 $\wedge$ `register7[0]` is 1)*>]>

3.3.1 Discussion

As the specification language can be considered as a specification language for describing a microcontroller. We discuss the specification language's precision, organization, and content completeness as we consider these important criteria.

The precision of the specification language for hardware knowledge

We discuss the precision of the specification language to examine whether the specification language is unambiguous [45]. As the proposed specification language describes hardware with two parts (i.e., a list of registers and a list of coding rules), we will discuss the precision of the language in describing these two parts. As fixed formats are provided for each part, there is no chance to provide an expression with multiple meanings. For the coding rule part, temporal properties are described using formal mathematical logic (i.e., PLTL formula), and other expressions are specified in fixed formats; this part has no ambiguity.

The organization of the specification language for hardware knowledge

The organization criteria examine whether a specification language is easy to understand and use [45]. As the proposed specification language is close to the natural language, the programmer can express their desired properties naturally. Although the temporal properties in this specification language are expressed in PLTL formulas, these formulas are represented by natural language terms (i.e., *before*, *after*, *before-or-after*). Hence, users without knowledge of PLTL operators can still read and write a specification in this specification language. To understand and use this specification language, only basic knowledge about registers and

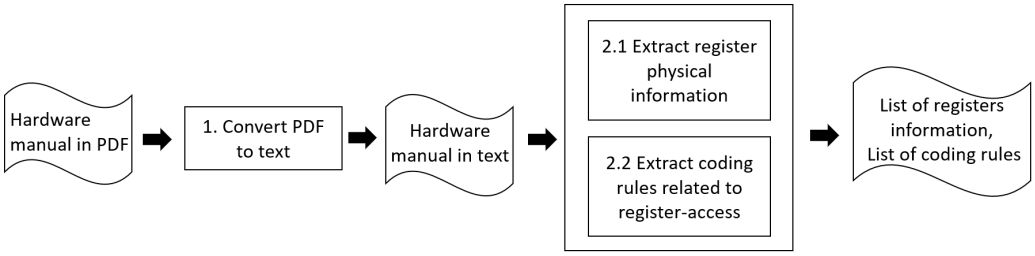

Figure 3.3: Approach for automatically analyzing hardware manual

microcontrollers is required. As the target users of the specification language are developers of embedded systems, this specification language is suitable for our purpose.

The content completeness of the specification language for hardware knowledge

The content completeness criteria examine whether the specification language is comprehensive to describe properties in the domain [45]. All notations in the specification language are necessary to specify coding rules belonging to defined categories in Table 3.1. Hence, the specification language is sufficient to cover the properties of our problem.

3.4 Automatically analyzing hardware manual

One of the difficulties in manually analyzing the hardware manual is the volume of the hardware manual. The target manual consists of 2415 pages and 641 register groups with 12,149 registers. Automating the extraction process will help to reduce the burden of manual tasks.

As discussed previously, hardware manuals are semi-structured in PDF. Physical information of registers is usually described in tables – which are structured formats. Coding rules of using register are described in normal sentences - which are non-structured formats. The idea to automatically analyze is first to convert the PDF file to plain text format; secondly, to extract physical information of registers by applying heuristics based on the structures of tables for the target information; lastly, to extract the target coding rules by applying heuristics based on keywords and syntactic features of the documents.

Figure 3.3 shows the proposed process for analyzing hardware manuals and extracting information for constructing formalized hardware knowledge. The input of this process is a hardware manual in PDF format. The output is a list of extracted registers and a list of coding rules.

There are two main phases in this process. In the first phase, the PDF file is converted to text. In the hardware manual, tables are usually used for describing important information. Especially fragments in the document that show the register's physical information are usually described in fixed formats in the table style. The table structure may present some semantics of the table data, like the relations among data components. The difficulty in converting PDF to text is that the solution should preserve to avoid the loss of information. In this research, we tried to use several settings to convert PDF documents to text and then select the best setting for our target. Specifically, we tried three settings. The first setting uses pdftotext [1] - a PDF text extraction implemented in python. The second setting is using Soda PDF [2] - a tool for opening, viewing, creating, converting, and editing PDF files - to convert PDF hardware manual to text. The third setting is to apply Soda PDF with Microsoft Word [3]. Specifically, Soda PDF is applied to convert PDF documents to word documents. Microsoft Word is used to convert word documents to text. In our experiment, we found that using pdftotxt, the format of the tables is not preferred well. Using Soda PDF, only the table format is preserved, but the information is presented, which is hard for later processing. Using Soda PDF and MS Word, even though the table format is not well visualized, the presentation of the information is more manageable for later processing. Finally, we choose the combination of Soda PDF with Microsoft Word as this is the best option among the three options to be easier to process later.

The second phase is for extracting information. The input in this phase is the hardware manual in text. The output is a list of extracted registers and a list of coding rules. There are two steps in this phase. In the first step, we extract physical information about the registers. The second step is to extract the coding rule related to register-access. The detail of this phase is shown in Section 3.4.1.

[1] https://pypi.org/project/pdftotext/

[2] https://www.sodapdf.com/

[3] https://www.microsoft.com/en-ww/microsoft-365/word?market=af

3.4.1 Extract information

Algorithm 1 Algorithm for extracting register information and coding rules related to register-access

1: **Input:** `HardwareManualInText`, `ListOfPattern`
2: **Output:** `ListOfRegisterObj`, `ListOfCodingRules`
3: **Procedure**:
4: // 1. Extract register physical information
5: `ListOfRegisterObj` = getRegisterObjs(`HardwareManualInText`, `ListOfPattern`[1-12])
6: // 2. Extract register-access coding rules
7: `ListOfCodingRules` = getCodingRules(`HardwareManualInText`, `ListOfPattern`[13-14])
8: return `ListOfRegisterObj`, `ListOfCodingRules`

This section handles the tasks of extracting the physical information and coding rules of registers. The approach for performing these tasks is to apply a heuristic based on the observation of the manual. Specifically, patterns in the regular expression are proposed to extract the target information. Table 3.2 summarizes the patterns used in this section. These are 14 patterns in total for different tasks. Pattern1 and Pattern2 for identifying register sections; Pattern3 is for identifying the areas of physical information; Pattern4 and Pattern5 are for extracting access sizes; Pattern6 for extracting the addresses; Pattern7 and Pattern8 are for extracting values after reset; Pattern9, Pattern10, and Pattern11 are for extracting the bit names; Pattern12 is for extracting the bit accessibilities; Pattern13 and Pattern14 are for extracting the coding rules related to written values. The usage of each pattern will be explained along each step in analyzing the hardware manual.

Algorithm 1 shows the procedure for extracting hardware knowledge. The input of this algorithm is `HardwareManualInText` - the hardware manual in text, `ListOfPattern` - patterns in Table 3.2. The output is texttt`ListOfRegisterObj` are `ListOfCodingRules`. There are two steps. The first step is identifying the register subsections and extracting physical information about the registers. Details of this step are presented in Section 3.4.1. The second step is extracting coding rules related to register-access. This step is discussed in Section 3.4.1.

Table 3.2: Patterns for extract register information

Task	Pattern	Name
Identify register sections	^[0-9.()]+[ ]+([a-zA-Z][a-zA-Z0-9 \/_<>]+)[ ]*—[ ]*(.*))$	Pattern1
	^[0-9]+.[0-9.]+[0-9][ ]+([a-zA-Z][a-zA-Z0-9\/_<>]+)([^—\n]*)Register	Pattern2
Identify areas of physical info	(.+?(?=Access:))(.+?(?=Address:))(.+?(?=Value after reset:|Value after power-on:))(.+?(?=Bit))(.+?(?=Note|Table|When))(.*)	Pattern3
Extract access size	[:].+?(?=(1-|8-|16-|32-)).+?(?=\.|$|(1-|8-|16-|32-))	Pattern4
	Access:(.*)[\t]*These registers.+?(?=(1-|8-|16-|32-)).+?(?=\.|$|(1-|8-|16-|32-))	Pattern5
Extract access address	Address:(.*)	Pattern6
Extract value after reset	([0-9A-Z x]+?(?=H))	Pattern7
	REGISTER_NAME: ([0-9A-Zx]+?(?=H))	Pattern8
Extract bit name	(.+?(?=Value after reset :|Value after power-on :))	Pattern9
	(.+?(?=Value after reset))(.+?(?=MARK))(.+?(?=Value after reset))	Pattern10
	(.*)\[([0-9]+):([0-9]+)\]	Pattern11
Extract bit accessibility	(R | R\/W | W | R*[0-9] | R\/W*[0-9] | W*[0-9] | R*W*)	Pattern12
Extract written value coding rules	(Set |Specify |Do not)(.+?)(?= to) to ([0-9]*[BH]|0|1|0.|1.|0|1) (in|to) (.*)	Pattern13
	(.*?)(?=The |This |Thus)(.+?)(?=should |should not |must |must not)(.+?)(?= to) to (([0-9]*[BH]|0 |1 |0.|1.) or ([0-9]*[BH]|0 |1 |0.|1.|0|1)|([0-9]*[BH]|0 |1 |0.|1.|0|1))	Pattern14

2.9.2.1 `Register1 / Register2` — Port Mode Control Register
2.13.4.1 `Register3_<name>` — Filter Control Register
3.3.3.4 `Register4` — Interrupt Function Setting
28.2.1.1 `Register15` Register

Figure 3.4: Examples of titles for register sections

Identifying register information subsections and extract physical information of registers based on heuristic

Each section of the hardware manual contains subsections for describing the section's related registers. Each subsection usually describes a group of registers. The titles of these sections and register information are usually described in several common formats as examples of several microcontrollers in Figure 3.2.

The manual of the target microcontroller also has a common format for the register section titles. Figure 3.4 shows several examples of titles of registers in the target hardware manual.

We can see that the titles are described in common formats, including a section number, the abbreviation name of the register group, and sometimes a short description/ name of the registers. Based on our observation, the title of the subsection either contains "—"in the middle of the "register" keyword. However, some sections contain "register" but do not for describing the physical information of the register groups. For example, the subsection "2.7.4.3 Writing to the `Register6` Register" is used to describe the method to write to register `Register6`, not the information of the register.

Physical information of registers, including the register name, address, accessed size, the value after reset, and bit information, is important in creating formalized specifications. In subsections for describing registers, the physical information of registers is usually described in a structured format as shown in Figure 3.2.

Algorithm 2 Algorithm for identifying register sections and extracting register objects

```
1: Input: HardwareManualInText, ListOfPattern
2: Output: ListOfRegisterObj
3: Procedure:
4: // 1. Identify register subsections
5: listOfRegisterObj = []
6: listOfRegisterSectionTitle = match(ListOfPattern[1] | ListOfPattern[2],
   hardwareManualInText)
7: for i in range(0, listOfRegisterSectionTitle.len) do
8:    m1 = listOfRegisterSectionTitle[i]
9:    m2 = listOfRegisterSectionTitle[i + 1]
10:   if "\nAdress:" in HardwareManualInText[m1.endIndex : m2.startIndex]
   then
11:      // 2. Extract register physical information
12:      for m in match(ListOfPattern[3],
   HardwareManualInText[m1.endIndex : m2.startIndex]) do
13:         registerObject = new RegisterObj()
14:         registerObject.accessSize = match(ListOfPattern[4] |
   ListOfPattern[5], m.group(2))
15:         registerObject.accessAddress = match(ListOfPattern[6] |
   ListOfPattern[7], m.group(3))
16:         registerObject.valueAfterReset = match(ListOfPattern[8] |
   ListOfPattern[9], m.group(4))
17:         registerObject.bitName = match(ListOfPattern[10] |
   ListOfPattern[11], m.group(5))
18:         registerObject.bitAccessibility = match(ListOfPattern[12],
   m.group(5)) listOfRegisterObj.add(registerObject)
19:      end for
20:   end if
21: end for
22: return listOfRegisterObj
```

Algorithm 2 shows the procedure for extracting register subsections and register information. Heuristics can handle this task. The input of this algorithm is `HardwareManualInText` - a hardware manual in text, `ListOfPattern` - patterns 1-12 in Table 3.2. The output of this algorithm is `listOfRegisterObj` - a list of register objects with physical information, including the access size, address, value after reset, bit names, and bit accessibility. The procedure has two steps: identify the register sections and extract register physical information. In Algorithm 2,

lines 6-10 in this algorithm is for the first step; lines 12-18 is for the second step.

The first step is conducted based on our observations on sections about registers in the targeted manual. There are two conditions to consider a section as a register section. The first condition is that the section title matches one of the following patterns for register section titles: Pattern1 or Pattern2. Line 6 in the algorithm checks this condition. As for presenting a register in the Definition 3.3.1, the name and the address are needed. Hence, the second condition is that the section describes the address as for reg of the content of the subsection contains "\nAddress:" which is the fixed format to describe the register address. Line 10 check whether the content among two titles found in line 6 contains this keyword. If yes, this is the register section.

Heuristics can handle the second step too. Based on our observations on subsections about registers, we create Patterns3 for detecting the register's physical information. Pattern3 has five matched groups for finding register size, register address, the value after reset, bit names, and bit accessibility. Line 12 applies Pattern3 to find fragments in the area between the pairs of closest register titles found in the first step.

Next, at line 14, we get the accessed sizes in the area containing characters that match group 2 of Pattern3. Three cases can occur. The first case is that if "Accessing from the user program is prohibited." or "This register is not accessible." appears in the accessed size area, the register group is not accessible by software. The second case is that if the accessed size is the same for all registers in the group, the accessed size will match group 1 of Pattern4. For example, *"Access:\tThis register can be read or written in 16-bit units."* matches the pattern. The third case is that if the accessed sizes are different for registers in the group, the accessed size will match group 1 of Pattern5. For example, *"Access:\tREGISTERNAME register can be read or written in 32-bit units. REGISTERNAME1 register can be read or written in 16-bit units. REGISTERNAME2 register can be read or written in 8-bit units."* matches the pattern.

The next step is at line 15 to get the accessed address in the area containing characters that match group 3 of Pattern3. If the text in the area is "Address: —", it means the register group is not accessible by software. Otherwise, we need to apply Pattern6 or Pattern7 over the matched group 3 of Pattern3. If the address match group 1 of Pattern6, the accessed address or address formula is the same for all registers in the group. For example, *"Address:\t<REGISTERNAME_base> + 44H"* matches the pattern. If the accessed address match group 1 of Pattern7, the accessed addresses or address formulas are different for registers in the group. For example, *"Address: REGISTERNAME1n: <REGISTERNAME1n_base> + 0300H + n × 4 (n = 0, 1, 2, 8, 9, 10, 11, 12, 18, 20) REGISTERNAME2n: <REGISTERNAME2n_base> + 03C8H + n × 4 (n = 0, 1) REGISTERNAME3: <REGISTERNAME3_base> + 0030H*1"* matches the pattern.

The next step is at line 16 to get the value after reset in the area containing characters that match group 4 of Pattern3. We apply Pattern8 and Pattern9 over this matched group. If the value after reset is the same for all registers in the group, the value after reset will march the group 1 of Pattern8. For example, *"Value after reset:\t00H"* matches the pattern. If the value after reset is different for registers in the group, the value after reset will match group 1 of Pattern9. For example, *"Value after reset: \tREGISTER0: FFFF FFFFH REGISTER20: FFFF FFFFH REGISTER30: FFFF FFFFH"* matches the pattern.

The next step at line 17 is to set the bit name in the area containing characters that match group 5 of Pattern3. We apply Pattern10 and Pattern11 over this matched group. If the accessed size is 8 or 16, the bit name will match group 1 of Pattern10. If the accessed size is 32, the bit name will match the combination of groups 1 and 3 of Pattern11.

The final step at line 18 is to get bit accessibility (readable/ writeable) in the area containing characters that match group 5 of Pattern3. We apply Pattern12 over this matched group. The bit accessibility will match group 1 of Pattern12.

Extracting coding rules using registers

Currently, we only target non-conditional coding rules. There are three groups of coding rules from the viewpoint of the information needed to extract. The first group contains coding rules related to the access size of the register. An example of this group is that *"This register can be read or written in 32-bit units."*. The second group is coding rules related to accessibility, including readability and writeability of individual bits in the registers. An example of this group is that *"Bit 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 R R R R R/W R/W R/W R/W R/W R/W R/W R/W R/W R/W R/W R/W R/W"*. The third group of coding rules is coding rules related to written values. An example of this group is that *"The BITNAME bit in REGISTERNAME register should be set to 0000B or 1111B ."*

Coding rules are written in the free format using natural languages. There are many possible ways to express the coding rules. However, the coding rules usually contain some special keywords such as "must", "must not", "should", "should not".

The general idea is to use heuristics based on the syntactic and linguistic features of the document. Specifically, the process of extracting the coding rules consists of four steps. The first step is preprocessing. In this step, we first break the documents into sentences using "." as the splitter. Subsequently, we apply the following techniques for each sentence: tokenizing, stemming, and part-of-speech tagging. In the second step, we apply Pattern13 and Pattern14 to get potential coding rules sentences: For example, "The BITNAME bit in REGISTERNAME register should be set to 0000B or 1111B ." matches Pattern14. In the third step, we obtain "NN", "NNP" tokens in the area containing characters that match

group 2 of Pattern13 and Pattern14. Lastly, among "NN" and "NNP" tokens, we check whether they are a bit name or register name. For example, BITNAME and REGISTERNAME are tagged as NNP and found as a bit name and a register name.

Chapter 4

Analyzing microcontroller-based software

4.1 Approach overview

Analyzing the source code and extracting knowledge related to register-access is essential for verifying the source code against the microcontroller-specific coding rules. For example, Figure 4.2 shows a microcontroller-based source code. In this program, write-access operators are performed at lines `19`, `24`, and `26`. We can check for coding rules related to write-access to registers at addresses `0x020E0004`, `0x020E0008`, `0x020E000C`, `0x020E0010`, and `0xFFF20300` if we know the information of these write-accesses. Knowledge related to register-access includes register-access expressions (i.e., locations in the source code where the accesses are performed), register-access details (i.e., accessed registers, accessed sizes, written values, or accessed bits), relations among register-access expressions (i.e., execution orders between register-access expressions).

However, there are two difficulties in analyzing the C program for extracting the target knowledge. The first difficulty comes from the large number of variations of the coding rules, which can be several hundred. Additionally, the coding rules can be updated, or new coding rules can be introduced. Hence, if a solution is not flexible enough, it will soon become useless. The second difficulty comes from the size and complexity of the systems. Nowadays, embedded systems can be very large with complex functionalities implemented. Hence, the solution should be sophisticated enough to handle large and complex source code.

There is an additional requirement that the target knowledge contains both syntactic such as the location of register-accesses in the source code, and semantic features of the systems, such as the written value in a write-access. Sallow analysis, like text-based analysis, is effective for syntactic properties; however, deep system

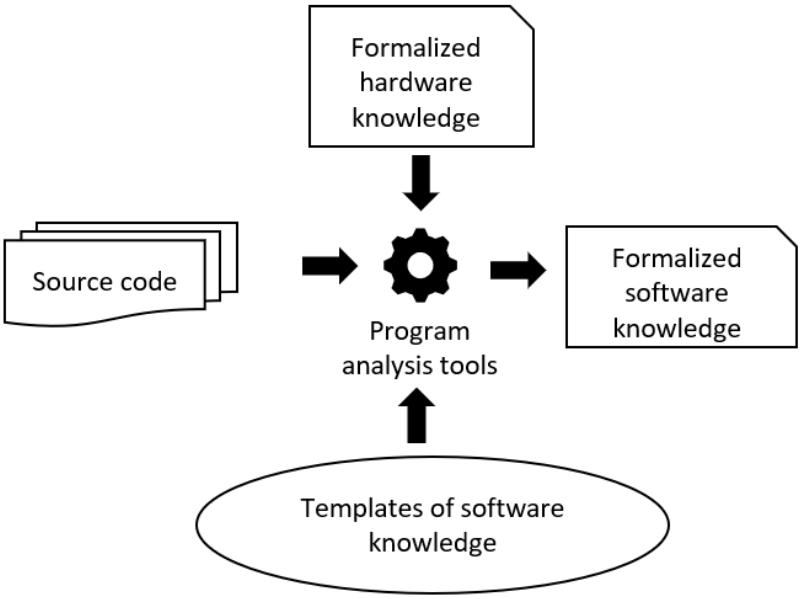

Figure 4.1: Overview of the process for analyzing C source code

analysis, such as pointer analysis, is required for syntactic properties. Hence, the solution should be effective in extracting both types of knowledge.

As it is difficult for a single technique to be both flexible and sophisticated, integrating multiple techniques with different strengths and weaknesses is a promising approach. We propose an algorithm combining four well-established program analysis techniques to extract the behaviors of source code. These program analysis techniques include PM, AI, BMC, and CEGAR. First, PM is employed in the form of code patterns to detect potentially related expressions of coding rules in the target source code. Secondly, AI, BMC, and CEGAR are employed to examine whether the potentially related expressions relate to or violate the coding rules.

This chapter first compares the four selected techniques in Section 4.2. Then, we explain the process for analyzing the source code using the algorithm which combines the selected techniques. Figure 4.1 shows an overview of this process. The input of this process is *Source code* - a C source code, *Formalized hardware knowledge* - the formalized hardware knowledge explained in the previous chapter, and *Specification language of software knowledge*. Similar to the *Specification language of hardware knowledge* discussed in Chapter 3, the *specification language of software knowledge* is proposed to precisely describe the extracted knowledge and enable the automated verification process. The definition of the *specification language of software knowledge* is explained in Section 4.3. The output is the formalized information, which is extracted from this source code, called *Formalized software knowledge*. Details of the algorithm to combine the four programming analysis techniques for extracting the target knowledge are shown in Section 4.4.

```
1  typedef struct struct_reg_long_data {
2      unsigned long address;
3      unsigned long value;
4  } StRegLongData;
5  const StRegLongData st_type01_02_0E[4] = {
6      { 0x020E0004, 0xF7317BDE},
7      { 0x020E0008, 0xF7313BDE},
8      { 0x020E000C, 0xFFFF5A5A},
9      { 0x020E0010, 0xFFF05A5A}
10 };
11
12 static void main(){
13     int cnt;
14     const StRegLongData *pt_st;
15     unsigned long mask = 0x4;
16
17     pt_st = &st_type01_02_0E[0];
18     for(cnt = 0 ; cnt < 4 ; cnt++){
19         (*((unsigned long*)pt_st->address)) = (unsigned long)
               pt_st->value;
20         pt_st++;
21     }
22
23     unsigned long* register_1 = (unsigned long*) 0xFFF20300;
24     *register_1 = 0x00000010;
25     while(*register_1 & mask != 0) {
26         *register_1 = foo();
27     }
28
29     return;
30 }
```

Figure 4.2: Motivation example for extracting knowledge from C program

As stated in Section 1.2, the target source code in this research is the C programs; assembly code is not taken into account. The second assumption is that we do not consider multiple thread systems and interruptions. The proposed approach in this chapter was published in [46, 47].

4.2 Comparison of selected techniques

Each selected technique has different strengths and weaknesses. Table 4.1 summarizes these techniques in two aspects: the well-suitedness and the employed approximation approach. For the first aspect, PM is good at handling syntactic properties; AI, BMC, and CEGAR can handle semantic properties; BMC and CE-

Table 4.1: Comparison of selected program analysis techniques

	Well-suitedness			Approximation approach	
	syntactic properties	semantic properties	explanation	over	under
PM	yes	no	no	no	no
AI	no	yes	no	yes	no
BMC	no	yes	yes	no	yes
CEGAR	no	yes	yes	yes	yes

GAR can provide explanations for detected errors. Techniques that target analyzing syntactic properties are often lightweight and flexible but imprecise. Techniques that target at analyzing semantic properties are often sophisticated but require a great amount of time and memory. For the second aspect, PM does not follow any particular approximation approach; AI employs the over-approximation; BMC follows the under-approximation; CEGAR implements both over- (the abstraction) and under-approximation (the refinement). Techniques which follow the over-approximation approach are sound but may generate false warnings. Techniques that follow under-approximation do not report false warnings but may miss actual bugs. CEGAR is an exception where both approximation approaches are implemented. Hence, this technique is sound and does not generate false warnings. However, because of multiple abstraction and refinement steps, CEGAR may consume a great amount of time and memory.

These four techniques can be combined to effectively extract knowledge related to register-accesses in source code. This knowledge includes syntactic properties (e.g., locations of the accesses in source code), semantic properties (e.g., accessed sizes of the accesses), and explanations for the accesses. Specifically, PM is employed to handle syntactic properties; AI and BMC are used to extract semantic properties; BMC and CEGAR are in charge of explaining the accesses. CEGAR is also good at handling semantic properties. However, after considering its high cost, we select CEGAR to handle the task of explaining only. Thanks to the different approximation approach employed, the selected techniques can complement each other to fulfill our target.

4.3 Formalization of software knowledge

This section defines the knowledge to be extracted from the source code. The first obvious knowledge for verifying systems against the target coding rules is knowledge related to register-access. In addition, handling loops is one technical

difficulty in analyzing microcontroller-based systems. In these systems, loops are frequently used to access a list of registers. For example, the `for` loop starting at line `18` in Figure 4.2 is used to access four registers at addresses `0x020E0004`, `0x020E0008`, `0x020E000C`, and `0x020E0010`.

In many cases, we need to know the number of loop iterations to decide whether an expression inside the loop access a register. Loop unwinding, in which a loop is unrolled with a fixed number of iterations, is the most intuitive and simple way to handle a loop. This technique can be easily applied to every loop in the program, even when we have little information about the target program. The disadvantage of this technique is that if the number of unrolled iterations is less than the actual number, this technique is unsound (i.e., a property is not ensured to hold outside of the bound). Loop unwinding can be applied with other program analysis techniques such as AI and BMC. However, the bound is normally a guessed number as there is usually insufficient information to make this selection.

Additionally, unrolling all loops in the source code can be expensive but unnecessary sometimes. For example, in Figure 4.2, the number of iterations of the `for` loop starting from line `18` decides which registers are accessed at line `19`. The loop starting from line `25` is irrelevant to the access at line `19`. Hence, unrolling the loop starting from line `18` only is a resource-efficient approach to checking the target access.

In this research, we extract the maximum number of loop iterations beforehand. This knowledge can support the process of extracting knowledge related to register-access. To sum up, there are two kinds of knowledge to be extracted: knowledge related to loops and knowledge related to register-access. We define software knowledge as a triple tuple as follows.

Definition 4.3.1 (Software knowledge) *A software knowledge **SWK** is a tuple **SWK** = <**name**, **listOfLoopObjects**, **listOfRegisterAccessObjects**>, where*

***name** ∈ **S**; **S** is the set of strings;*
***listOfLoopObjects** = [**loop_info_1**, .., **loop_info_n**]; **n** ∈ ℕ∗;*
***listOfRegisterAccessObjects** = [**access_info_1**, .., **access_info_n**], **n** ∈ ℕ∗;*
***loop_info_1** ∈ **SFL**, **SFL** is a set of formalized loops;*
***access_info_1** ∈ **SFRA**, **SFRA** is a set of formalized loops;*

A source code is represented by a **name** - the name of the source code, a **listOfLoopObjects**, and a **listOfRegisterAccessObjects**. Objects in **listOfLoopObjects** are described in the specification language shown in Section 4.3.1. Similarly, objects in **listOfRegisterAccessObjects** follow the specification language in Section 4.3.2. As the loop objects and register-access objects are extracted using program analysis techniques/ tools, the output formats of these techniques/ tools are considered for defining the formalization of these objects.

4.3.1 Specification language for loop object

We define a loop object as the tuple of **id**, *syntacticInfo*, *loopType*, and *maxNumOfIteration* as follow.

Definition 4.3.2 (Loop object) *A loop L is a tuple L = <**id**, **syntacticInfo**, **loopType**, **maxNumOfIteration**>, where*

id $\in \mathbb{N}$,

syntacticInfo *= <**fName**, **funcName**, **lineNumberStart**, **lineNumberEnd**>;*
fName $\in$ *FNAME;* ***funcName*** $\in$ *FUNCNAME;*
lineNumberStart, ***lineNumberEnd*** $\in$ *[0..MAXLINE-1];*
lineNumberEnd *>= **lineNumberStart**;*

loopType $\in$ *{`for`, `while`};*
maxNumOfIteration $\in$ *{ val,* `unknown`*}, val* $\in \mathbb{N}$*;*

FNAME is a set of names of files in source code;
*FUNCNAME is a set of names of functions in a file named **fName**;*
*MAXLINE is the maximum line number in a file named **fName**.*

Each loop includes an **id** for identifying the object, a **syntacticInfo** to locate the loop, a **loopType** to specify the type of loops, and a **maxNumOfIteration** to store the maximum number of iterations. There are two types of loops handled: `for` and `while`. In C language, there are several methods for constructing loops: `for` statements, `while` statements, `goto` statements, and recursive functions. As using `goto` statements and recursive functions are not recommended in practice, we only handle `for` statements and `while` statements. The **maxNumOfIteration** may not be available for all `for` and `while` loops. At this moment, only loops in which the executing condition is controlled by one variable and the value of this variable is updated by a constant after each iteration are supported.

For example, the `for` loop starting at line `18` of Figure 4.2 is supported. However, the `while` loop starting at line `25` is not supported as this loop is controlled by the return value of function `foo`, which is outside of the loop. If the maximum number of iterations of the loop beginning at line `18` is calculated, this loop will be represented as `<0, <main.c, main, 18, 21, for, 4>`. As the loop beginning at line `25` is not supported, the representation for this loop is `<1, <main.c, main, 25, 27>, while, unknown>`.

4.3.2 Specification language for register-access object

We define a register-access object as the tuple of **id**, **syntacticKnowledge**, **semanticKnowledge** as follows.

Definition 4.3.3 (Register-access object) *A register-access RegA is a tuple RegA = <**id**, **syntacticKnowledge**, **semanticKnowledge**>, where*

id $\in \mathbb{N}$*;*

syntacticKnowledge $=$ *<* ***locationInfo***, ***accessType***>*;*
locationInfo $=$ *<**fileName**, **functionName**, **lineNumberStart**, **lineNumberEnd**>;*
fileName $\in$ *FNAME ,* ***functionName*** $\in$ *FUNCNAME;*
lineNumberStart; ***lineNumberEnd*** $\in$ *[0..MAXLINE-1];* ***lineNumberEnd*** $>=$ ***lineNumberStart****;*
accessType $\in$ *{read-access, write-access};*

semanticKnowledge $=$ *<* ***registerName***, ***accessDetail***, ***relations***, *<**status**, **precision**>*, ***explanation***>*;*
registerName $\in$ *REGISTER_NAME_LIST;*
REGISTER_NAME_LIST is the list of register names in the used hardware;

$$\textbf{\textit{accessDetail}} = \begin{cases} < \textbf{\textit{accessedBits}}, \textbf{\textit{accessedSize}} > \ for\ read\text{-}access, \\ < \textbf{\textit{writtenValue}}, \textbf{\textit{accessedSize}} > \ for\ write\text{-}access; \end{cases}$$

accessedBits $=$ ***integerSet****;*
writtenValue $=$ ***integerSet****;*

$$\textbf{\textit{integerSet}} = \begin{cases} \textbf{\textit{interval}}; \textbf{\textit{interval}} =< l, u >; l, u \in \mathbb{N}; \\ \textbf{\textit{periodicityInterval}}; \textbf{\textit{periodicityInterval}} =< l, u, m, r >; l, u, m, r \in \mathbb{N}; \\ \textbf{\textit{enum}}; \textbf{\textit{enum}} = [n_0, .., n_i]; n_i =< val, \textbf{\textit{precision}}, \textbf{\textit{explanation}} >; \end{cases}$$

$val \in \mathbb{N}$*;*
accessedSize $\in$ *{8, 16, 32, 64};*
relations $=$ *{<**id**, <**executionOrder**, **precision**, **explanation**> > |* ***id*** $\in \mathbb{N}$*};*
executionOrder $\in$ *{before, after};*
status $=$ *{accessed, notAccessed, dead};*
precision $=$ *{confirmed, maybe, unknown};*
explanation $\in$ *{**executionSequence**$_n$, notAvailable},* $n \in \mathbb{N}$*;*
executionSequence$_n$ $=$ *{<*s_0*.**threadNum**,* s_0*.**lineNum**,* s_0*.**fileName**>, .. , <*s_i*.**threadNum**,* s_i*.**lineNum**,* s_i*.**fileName**>, .. <*s_n*.**threadNum**,* s_n*.**lineNum**,* s_n*.**fileName**>};*
s_0 *is the program entry point;* ***threadNum*** $\in \mathbb{N}$*;* ***lineNum*** $\in \mathbb{N}$*;* ***fileName*** $\in$ *FNAME;*
FNAME is a set of names of files in source code;
FUNCNAME is a set of names of functions in a file named ***fileName****;*

*MAXLINE is the maximum line number in a file named **fileName**.*

Each register-access contains an **id** and two kinds of knowledge: **syntacticKnowledge** and **semanticKnowledge**. The **syntacticKnowledge** includes the **locationInfo** in the source code and the **accessType**. The **locationInfo** of a register-access is presented by the **fileName**, **functionName**, **lineNumberStart** and **lineNumberEnd** to present where the access is performed. The **accessType** may be *read-access* or *write-access*. We consider the access type an element of **syntacticKnowledge** as *read-access* and *write-access* can be identified by examining the syntactical features of the source code. For example, if a C expression contains an assignment operator, it is a *write-access*. For example, the **syntacticKnowledge** for the *write-access* at line `19` of Figure 4.2 is as follow.

< <main.c, main, 19, 19>, *write-access*>

The **semanticKnowledge** includes the accessed **registerName**, an **accessDetail** which is accessed size and bits for *read-access* and accessed size and written value for *write-access*, **relations** with other accesses, which are the execution order among these accesses, an **status** which is either *accessed*, *notAccessed*, or *dead*, and an **explanation** which is an execution trace from the beginning of the program to the accessed point.

In the **accessDetail**, the **accessBits**, and **writternValue** are represented by an **integerSet**. The **accessBits** and **writternValue** can be calculated by checking the values of C expressions. For example, the written value of the *write-access* at line `19` of Figure 4.2 can be obtained by calculating the values of the expression `(unsigned long)pt_st->value`. Another example is the accessed bits of the *read-access* at line `26`. In this example, the accessed bits can be obtained by calculating the value of the expression `mask`. Using AI, BMC, and CEGAR, the value of these expressions (`(unsigned long)pt_st->value` and `mask`) can be calculated by checking the over-approximated value set or checking whether the expression has a particular value. Hence, the calculated value can be a single value or a set of possible values. Hence, we use a **integerSet** for presenting these values. Inspired by the integer set representation of the Eva plugin of Frama-C[1], we represent a **integerSet** in one of three formats: **interval**, **periodicityInterval**, and **enum**. A **interval** <l, u> represents all the integers between l and u, including the bound. For example, <1, 3> is the set of {1, 2, 3}. A **periodicityInterval** <l, u, r, m> represents the set of values between l and u whose remainder in the Euclidean division by m is equal to r. For example, <2,42,2,10> represents the set that contains 2, 12, 22, 32, and 42. A **enum** represents an integer set by listing the

[1]https://frama-c.com/fc-plugins/eva.html

value one by one. However, the value in this set is a triple tuple of *val*, **precision**, and **explanation**. The *val* is the value itself, and the **precision** represents the certainty level of whether the value is actually used in at least one reachable execution path of the program. There are three levels of certainty: *confirmed* - 100% confident, *maybe* - less than 100% confident, and *unknown* - we do not have any information. The **accessSize** represents the size of the access which can be either `8`, `16`, `32`, or `64`.

The **relation** represents the temporal properties among the register-access object and other register-access objects in the source code. A **relations** is the list of double tuples. A tuple has an **id** - id of another register-access object and a tuple of **executionOrder**, **precision**, and **explanation**. The value of **executionOrder** can be either *before* or *after*. Similar to the **precision** in the **enum**, the **precision** here also represents the level of certainty of the **executionOrder**. The **explanation** represents a sequence of states to show that the order is reachable in the execution.

The **status** of a register-access is the answer to a yes/no question: whether the specific expression accesses the particular register. For example, an access status can answer the following question: Does the expression at line `19` in Figure 4.2 access register at the address `0x020E0008`? The answer to this question may fall into three cases: yes (`accessed`) - this is a register-access; reachable and no (`notAccessed`) - this is not a register-access; unreachable (`dead`) - the expression is not reachable.

The **precision** in a register-access is decided based on the approximation approaches (i.e., over and under) implemented in the employed program analysis techniques. Inspired by the Kripke structure used in model checking [48], a Kripke structure is defined to represent the actual behavior, and two Kripke structures are defined to represent the approximated behaviors of the analyzed program. Let A, A_{under}, A_{over} be sets of atomic propositions about register-access where $A_{under} \subseteq A \subseteq A_{over}$. $K = <S, R, L>$ be a Kripke structure over A for representing the concrete behavior of the analyzed program; $K_{under} = <S_{under}, R_{under}, L_{under}>$ be a Kripke structure over A_{under} for representing an under-approximated behavior of the analyzed program; $K_{over} = <S_{over}, R_{over}, L_{over}>$ be a Kripke structure over A_{over} for representing an over-approximated behavior of the analyzed program. S, S_{under}, and S_{over} be finite sets of states so that $S_{under} \subseteq S \subseteq S_{over}$. $R \subseteq S \times S$, $R_{under} \subseteq S_{under} \times S_{under}$, and $R_{over} \subseteq S_{over} \times S_{over}$ be sets of transitions. L: $S \rightarrow 2^A$, L_{under}: $S_{under} \rightarrow 2^{A_{under}}$, and L_{over}: $S_{over} \rightarrow 2^{A_{over}}$ be functions which map each state to a set of atomic propositions about register-access. Let *ra* be a state where a register-access is performed, p be an atomic proposition that represents that *ra* is reachable. Let *s0* be the entry point of the analyzed program, and the five possible statuses of a register-access *ra* are defined in CTL^* [49] as below:

Definition 4.3.4 (Register-access precision) *A register-access precision is one of*

the five values:

$$\texttt{confirmed} \iff K_{under}, s0 \models \boldsymbol{EF}p \text{ or } K_{over}, s0 \models \boldsymbol{AF}p$$

$$\texttt{confirmedNot} \iff K_{over}, s0 \nvDash \boldsymbol{EF}p$$

$$\texttt{maybe} \iff K_{over}, s0 \models \boldsymbol{EF}\ p$$

$$\texttt{dead} \iff \nexists s,\ s \in S_{over} \wedge s.syntacticInfo = ra.syntacticInfo$$

$$\texttt{unknown} \iff otherwise$$

There are five possible values of **precision** for a register-access object when analyzing the source code with the four program analysis techniques: `confirmed`, `confirmedNot`, `maybe`, `dead`, and `unknown`. However, only register- `confirmed`, `maybe`, `unknown`

The **explanation** provides evidence of the reachability of a register-access object. As briefly explained above, an **explanation** is an execution trace from the beginning of the program to the accessed point. The formal definition of an **explanation** is an ordered set of triple tuples in the form of <s_i.**threadNum**, s_i.**lineNum**, s_i.**fileName**> represents an execution point with a thread number, a line number, and a file name. Although currently, we only handle single thread programs, the **threadNum** is preferred for easily extending the approach to handle multiple thread programs. Now, **threadNum** is always `0`. The **explanation** is available for register-access objects detected using the under-approximation approach (i.e., BMC and CEGAR).

For example, assume that the register's name at address `0x020E0004` is `registerA`. If the write-access expression at line `19` to `registerA` in Figure 4.2 is confirmed by BMC, the **semanticKnowledge** will be represented as follows.

> <registerA, < <[<4147215326, *maybe*, *notAvailable*>, <4147198942, *maybe*, *notAvailable*>, <4294924890, *maybe*, *notAvailable*>, <4293941850, *maybe*, *notAvailable*>], 32>, {<1, <before, *maybe*, *notAvailable*> >, <2, <before, *maybe*, *notAvailable*> >, <3, <before, *maybe*, *notAvailable*> >, <4, <before, *maybe*, *notAvailable*> >}, <accessed, confirmed>, {<0, 1, main.c>, <0, 2, main.c>, <0, 3, main.c>, <0, 4, main.c>, <0, 5, main.c>, <0, 6, main.c>, <0, 7, main.c>, <0, 8, main.c>, <0, 9, main.c>, <0, 10, main.c>, <0, 11, main.c>, <0, 12, main.c>, <0, 13, main.c>, <0, 14, main.c>, <0, 15, main.c>, <0, 16, main.c>, <0, 17, main.c>, <0, 18, main.c>, <0, 19, main.c>} >

In which, registerA is the **registerName**; <[<4147215326, *maybe*, *notAvailable*>, <4147198942, *maybe*, *notAvailable*>, <4294924890, *maybe*, *notAvailable*>, <4293941850, *maybe*, *notAvailable*>], 32> is the **accessDetail**; {<1, <before, *maybe*, *notAvailable*> >, <2, <before, *maybe*, *notAvailable*> >, <3, <before, *maybe*, *notAvailable*> >, <4, <before, *maybe*, *notAvailable*> >} is the **relations**; <accessed, confirmed> is the <**status**, **precise**>; {<0, 1, main.c>, <0, 2, main.c>, <0, 3, main.c>, <0, 4, main.c>, <0, 5, main.c>, <0, 6, main.c>, <0, 7, main.c>, <0, 8, main.c>, <0, 9, main.c>, <0, 10, main.c>, <0, 11, main.c>, <0, 12, main.c>, <0, 13, main.c>, <0, 14, main.c>, <0, 15, main.c>, <0, 16, main.c>, <0, 17, main.c>, <0, 18, main.c>, <0, 19, main.c>} is the **explanation**.

The full formalization of the write-access at line 19 to registerA in Figure 4.2 is shown as follows.

- <0, < <main.c, main, 19, 19>, *write-access*> <registerA, < <[<4147215326, *maybe*, *notAvailable*>, <4147198942, *maybe*, *notAvailable*>, <4294924890, *maybe*, *notAvailable*>, <4293941850, *maybe*, *notAvailable*>], 32>, {<1, <before, *maybe*, *notAvailable*> >, <2, <before, *maybe*, *notAvailable*> >, <3, <before, *maybe*, *notAvailable*> >, <4, <before, *maybe*, *notAvailable*> >}, <accessed, confirmed>, {<0, 1, main.c>, <0, 2, main.c>, <0, 3, main.c>, <0, 4, main.c>, <0, 5, main.c>, <0, 6, main.c>, <0, 7, main.c>, <0, 8, main.c>, <0, 9, main.c>, <0, 10, main.c>, <0, 11, main.c>, <0, 12, main.c>, <0, 13, main.c>, <0, 14, main.c>, <0, 15, main.c>, <0, 16, main.c>, <0, 17, main.c>, <0, 18, main.c>, <0, 19, main.c>} > >

4.4 Automatically extracting source code knowledge

4.4.1 Approach overview

Algorithm 3 Extracting knowledge related to register-access

Input: SC, PatternRegAccess, ListOfRegs, PatternAllLoop, PatternSupportedLoop
Output: ListOfAccObjs
Procedure:
/ Step 1: Extract loop objects in source code */*
ListOfLoopObjs = extractLoopObj(SC, PatternAllLoop, PatternSupportedLoop)
/ Step 2: Extract register-access objects in source code */*
ListOfAccObjs = detectRegAccObj(SC, PatternRegAccess, ListOfRegs, ListOfLoopObjs)
/ Step 3: Extract access detail for register-access objects*/*
ListOfAccObjs.updateAccDetail(extractAccDetail(SC, ListOfLoopObj, ListOfAccObjs))
/ Step 4: Extract relations between register-access objects */*
ListOfAccObjs.updateAccRelation(extractRelations(SC, ListOfLoopObj, ListOfAccObjs))
return ListOfAccObjs

Details of the process to extract register-access knowledge are shown in Algorithm 3. The inputs of this process are SC - a target source code, PatternRegAccess - the code patterns for register-access which are shown in Definition 4.4.1, ListOfRegs - the list of registers in the target hardware, PatternAllLoop - the code patterns for for and while loops which are shown in Definition 4.4.2, PatternSupportedLoop - the code patterns for supported loops which are also shown in Definition 4.4.2. The PatternRegAccess, which is shown in Section 4.4.2, focuses on detecting several forms of pointer access as using a pointer is one of the most common methods for performing register-access [18]. The PatternAllLoop simply detects all for and while loops. The PatternSupportedLoop detects for and while loops where the executing condition is controlled by one variable, and the value of this variable is updated by a constant after each iteration.

The process consists of four steps. The first step is to extract knowledge related to loops in the target source code. The output of this step is the list of loop objects in the formalized form, which is shown in Section 4.3.1. The second step is to detect register-access objects in the source code. By the end of this step, a list of register-access objects is obtained. However, the information about access detail

and relations among these objects has not been available. The detail of this step can be found in Section 4.4.4. The next two steps extract the knowledge about access detail and relations among these objects. By the end of this process, we obtain `ListOfAccObjs` - a list of register-access objects in the formalized form, which is shown in Section 4.3.2.

4.4.2 Code patterns for register-access

Although a microcontroller-based application may have thousands of lines of code, only several fragments of this source code are related to a coding rule. Inspecting all lines is unnecessary and even costly. Narrowing down the focus on potentially related fragments is a resource-efficient solution. There is a concern about the precision in the case that the actually related fragments are not inspected. However, each company may have coding conventions for performing a specific task. When we need to verify whether a task is performed correctly, inspecting fragments, which follow these coding conventions for this task, is sufficient. Hence, a solution can be both effective in terms of consumed resources and precision if flexible enough to adjust for different coding styles easily.

In this research, we focus on the coding rules related to register-access and source code written in the C programming language. Register-access expressions are expressions in which the address of a register is accessed. This property distinguishes register-access expressions from normal access expressions, which also access an address (e.g., the address of a variable) but not a register address. In this research, the read-access and the write-access refer to register-access, not normal access. In [18], Saks explained several methods for register-access operators using the C language. As it is hard to cover all variations of register-access in C source code, we target only popular variations in this research. Additionally, companies usually have their conventions to perform register-access as this is a common task in embedded software. In this research, we discussed with developers of AISW the frequently used methods for register-access at their company. Based on these discussions, we proposed code patterns for covering these methods. As the code patterns are proposed based on the discussions with people of AISW only, one may argue that these patterns are biased to the methods used at this company. However, even if the code patterns only fit with the coding style at AISW, the proposed approach is flexible enough so that other register-access methods can be introduced easily as new patterns.

The code patterns for register-access are defined in BNF as follows. Differently from the original BNF, the definition is shown in a top-down style for easier following.

Definition 4.4.1 (Code patterns for register-access) *The code pattern for read-*

access operators is:

<assignment-operator> ::= "=" | "&=" | "|=" | "^=" | "+=" | "-=" | "=" | "/=" | "%=" | ">>=" | "<<="*
<pre-side-effect-oper> ::= "++" | "–"
<post-side-effect-oper> ::= "++" | "–"
<not-equal-sign> is any valid token in C program other than "=";
<expression> is a valid expression in C program;
<not-number-or-type> is not a number or a C primitive data type;
<read-access-pattern> ::= <pointer-access-expression> <not-equal-sign> **(1)**
| <pointer-access-expression> "&" <expression> **(2)**
| (<pointer-access-expression> "<<" <expression>) "&" <expression> **(3)**
| (<pointer-access-expression> ">>" <expression>) "&" <expression> **(4)**
| (<pointer-access-expression> "&" <expression>) ">>" <expression> **(5)**
| (<pointer-access-expression> "&" <expression>) ">>" <expression> **(6)**

The code pattern for write-access operators is:

<write-access-pattern> ::=
<pointer-access-expression> <assignment-operator> <expression> **(1)**
| <pre-side-effect-oper> <pointer-access-expression> **(2)**
| <pointer-access-expression> <post-side-effect-oper> **(3)**

<pointer-access-expression> ::= <not-number-or-type> "" <expression>* **(1)**
| <expression> "->" <expression> **(2)**
| (""<expression>) "." <expression>* **(3)**

These code patterns are proposed for checking the coding rules related to register-access. For read-access, code pattern number **(1)** is used to check the properties related to the readability of a whole register and the access size. This pattern will extract pointer-access expressions which match the *<pointer-access-expression>* pattern but not follow by the equal sign (i.e., "="). If an expression is followed by the equal sign, it is a write-access expression, not a read-access one. There are two expressions at line `13` (i.e., "`*register_3`" and "`*register_1`") and an expression at line `14` (i.e., "`*register_4`") of the source code in Figure 4.3 match this code pattern. Code pattern number **(2)**, **(3)**, **(4)**, **(5)**, and **(6)** are used to check the properties related to the readability of a bit in a register as bitmask is often used for reading specific bits. There are two expressions at line `13` (i.e., "`(*register_3 >> 4) & 1`" and "`(*register_1 >> 1) & 1)`" in Figure 4.3 match the read-access code pattern number `(4)`.

```
1  #define REGISTER_1_ADD    0xFFF20300
2  #define REGISTER_3_ADD    0xFFF20304
3  #define REGISTER_4_ADD    0xFFF20338
4  void func_write_32 (unsigned long add, unsigned long data){
5      (*((unsigned long*)add)) = (unsigned long)data;
6  }
7  void main(void) {
8      func_write_32(REGISTER_3_ADD, 0x00000000);
9      func_write_32(REGISTER_4_ADD, 0x00000000);
10     unsigned long* register_1 = (unsigned long*) REGISTER_1_ADD;
11     unsigned long* register_3 = (unsigned long*) REGISTER_3_ADD;
12     unsigned long* register_4 = (unsigned long*) REGISTER_4_ADD;
13     if (! ((*register_3 >> 4) & 1) & ((*register_1 >> 1) & 1))
14         *register_4 |= 0x00000010;
15     func_write_32(REGISTER_4_ADD, 0xEFFFFFFF);
16 }
```

Figure 4.3: Example program containing matched expressions of the code patterns for register-access

For the write-access, all three code patterns are used to check for properties related to the access size and the written values of a whole register or a bit in a register. The code pattern number **(1)** is an assignment that assigns a value of the <*expression*> in the right-hand side to the <*pointer-access-expression*> on the left-hand side. There are two expressions (at line 5 and line 14) in Figure 4.3 that match this code pattern. Code pattern for the write-access number **(2)** and **(3)** are used to detect expressions where a <*pointer-access-expression*> is modified using pre/post side effect operators. There is no expression in Figure 4.3 that matches these code patterns.

There are three code patterns for pointer-access. The code pattern number **(1)** is used to detect expressions that contain an asterisk sign which is followed by an expression. However, there must be no token which is a number or a primitive data type, before the asterisk sign. If there is a number before the asterisk sign, the asterisk sign is not the pointer sign but the multiplication operator. If there is a data type before the asterisk sign, it is a definition of a pointer variable but not our target (e.g., line number 10, 11 and 12 in Figure 4.3). There are four expressions in Figure 4.3 match with the code pattern **(1)** of pointer-access: "`*((unsigned long*)add)`" at line 5, `*register_3` and `*register_1` at line 13, `*register_4` at line 14. Code pattern **(2)** and **(3)** for pointer-access are semantically equivalent. They both are used to detect expressions that access elements of structures or unions. There is no expression in Figure 4.3 that matches these code patterns.

The code patterns are expressed using the Extended Backus-Naur Form (EBNF) [50]. In fact, the code patterns represent a subset of the syntax of the C programming language; BNF (and several context information) is also employed for expressing this syntax [51]. In this research, we employed Cobra for describing

code patterns and extracting expressions that match these code patterns.

This section described how to extract fragments of code that potentially relate to the coding rules. The next section will show how to evaluate these extracted fragments to know whether they actually relate to or violate the coding rules.

4.4.3 Algorithm to extract loop objects

Algorithm 4 Extracting knowledge related to loops

```
1:  Input: SC, PatternAllLoop, PatternSupportedLoop
2:  Output: ListOfLoopObjs
3:  Procedure:
4:  supportedLoops = []
5:  for exp in SC do
6:      (isMatched, loopInfo) = PM.match(exp, PatternSupportedLoop)
7:      if isMatched == TRUE then
8:          supportedLoops.add(loopInfo)
9:      else
10:         (isMatched, loopInfo) = PM.match(exp, PatternAllLoop)
11:         if isMatched == TRUE then ListOfLoopObjs.add(loopInfo, supported = FALSE)
12:         end if
13:     end if
14: end for
15: annotatedSC = attachAnnotationForCalLoopIter(SC, supportedLoops)
16: (loopObjs, numsOfIter) = AI.analyze(annotatedSC)
17: for i in [0, len(loopObjs)) do
18:     ListOfLoopObjs.add(loopObjs[i], numsOfIter[i], supported = TRUE)
19: end for
20: return ListOfLoopObjs
```

This section explains the procedure for extracting knowledge about loops in source code. Algorithm 4 shows detail of this process. The input of this process is `SC` - the target source code, `PatternAllLoop` - the code patterns for detecting all `for` and `while` loop, and `PatternSupportedLoop` - the code patterns for detecting supported loops which we can calculate the maximum number of iterations. The code patterns for loops are defined in BNF as follows. Differently from the original BNF, the definition is shown in a top-down style for easier following.

Definition 4.4.2 (Code patterns for loops) *The code pattern for all* `for` *and* `while` *loops is:*
<all-loop-pattern> ::= `for (.*) {.*} | while (.*) {.*}`

The code pattern for supported loops is:

<supported-loop-pattern> ::= <simple-for-pattern> | <simple-while-pattern>

<simple-for-pattern> ::= `for` *(<ident> = <const>; <ident> <compare-oper> <exp>; <update-exp>) { .* <update-exp>? .* }*
<simple-while-pattern> ::= `while` *(<ident> <compare-oper> <exp>) { .* <update-exp> .* }*

<update-exp> ::= <pre-side-effect-oper> <ident> | <ident> <post-side-effect-oper>
| <ident> <assignment-operator> <const> | <ident> = <ident> <operator> <const>
| <ident> = <const> <operator> <ident>
<compare-oper> := "<" | "<=" | ">" | ">=" | "!="
<operator> := "+" | "-" | "" | "/" | "%"*
<assignment-operator> := "^=" | "+=" | "-=" | "=" | "/=" | "%=" | "»=" | "«="*
<pre-side-effect-oper>, <post-side-effect-oper> := "++" | "- -"
<exp> is a valid expression in C program;

At this moment we only calculate the maximum number of iterations for simple loops. Definition 4.4.2 shows the `PatternAllLoop` and the `PatternSupportedLoop`. The `PatternAllLoop` detects all expressions with the keyword `for` or `while`. The `PatternSupportedLoop` detects simple `for` and `while` loops which are controlled by one identifier and the identifier is updated inside the loops only. The output of this process is `ListOfLoopObjs`. The format of a loop object is defined in Section 4.3.1. The extraction process is done by combining two program analysis techniques (i.e., PM and AI). Specifically, PM is in charge of detecting loop expressions in the source code. AI is in charge of calculating the values of these loop expressions so that we can decide the maximum number of iterations for these loops.

4.4.4 Algorithm to detect register-access objects

Algorithm 5 Extract register-access objects

```
1: Input: SC, PatternRegAcc, ListOfRegs, ListOfLoopObjs
2: Output: ListOfAccObjs
3: Procedure:
4: assertForm = ["NotEqual", "Equal"]
5: /* 1. Extract potential register-access expressions */
6: ListOfPotentialRegAcc = []
7: for exp in SC do
8:     (status, regAccInfo) = PM.match(exp, PatternRegAcc)
9:     if status == TRUE then ListOfPotentialRegAcc.add(regAccInfo) end if
10: end for
11:
12: /* 2. Extract actual register-access objects and provide explanations */
13: /* 2.1. AI-NotEqual-No loop unwind */
14: annotatedSC = attachAssertion(SC, ListOfPotentialRegAcc, ListOfReg, assertForm[0])
15: (confirmedNotObjs, maybeObjs, unknownObjs) = AI.analyze(annotatedSC)
16: for obj in maybeObjs do ListOfAccObjs.add(obj, "maybe") end for
17: for obj in unknownObjs do ListOfAccObjs.add(obj, "unknown") end for
18:
19: /* 2.2. AI-NotEqual-Loop unwind */
20: unknownObjs = []
21: for obj in ListOfAccObjs do
22:     if obj.status == "unknown" then unknownObjs.add(obj) end if
23: end for
24: annotatedSC = attachAssertion(SC, unknownObjs, assertForm[0], ListOfLoopObjs)
25: (confirmedNotObjs, maybeObjs, unknownObjs) = AI.analyze(annotatedSC)
26: for obj in maybeObj do ListOfAccObjs.update(obj, "maybe") end for
27: for obj in confirmedNotObjs do ListOfAccObjs.remove(obj) end for
28:
29: /* 2.3. AI-Equal-No loop unwind */
30: maybeObjs = []
31: for obj in ListOfAccObjs do
32:     if obj.status == "maybe" then maybeObjs.add(obj) end if
33: end for
34: annotatedSC = attachAssertion(SC, maybeObjs, assertForm[1])
35: (confirmedObjs, unknownObjs) = AI.analyze(annotatedSC)
36: for obj in confirmedObjs do ListOfAccObjs.update(obj, "confirmed") end for
```

```
37:
38: /* 2.4. AI-Equal-Loop unwind */
39: maybeObjs = []
40: for obj in ListOfAccObjs do
41:     if obj.status == "maybe" then maybeObjs.add(obj) end if
42: end for
43: annotatedSC = attachAssertion(SC, maybeObjs, assertForm[1], ListOfLoopObjs)
44: (confirmedObjs, unknownObjs) = AI.analyze(annotatedSC)
45: for obj in confirmedObjs do ListOfAccObjs.update(obj, "confirmed") end for
46:
47: /* 2.5. BMC-NotEqual-Loop unwind */
48: maybeUnknownObjs = []
49: for obj in ListOfAccObjs do
50:     if obj.status == "maybe" or obj.status == "unknown" then maybeUnknownObjs.add(obj)
    end if
51: end for
52: annotatedSC = attachAssertion(SC, maybeUnknownObjs, assertForm[0], ListOfLoopObjs)
53: ((confirmedObjs, explanations), unknownObjs) = BMC.analyze(annotatedSC)
54: for i in [0, len(confirmedObjs)] do
55:     ListOfAccObjs.update(confirmedObjs[i], "confirmed", explanations[i])
56: end for
57:
58: /* 2.6. BMC-Equal-Loop unwind */
59: maybeUnknownObjs = []
60: for obj in ListOfAccObjs do
61:     if obj.status == "maybe" or obj.status == "unknown" then maybeUnknownObjs.add(obj)
    end if
62: end for
63: annotatedSC = attachAssertion(SC, maybeUnknownObjs, assertForm[1], ListOfLoopObjs)
64: (confirmedObjs, unknownObjs) = BMC.analyze(annotatedSC)
65: for obj in confirmedObjs do ListOfAccObjs.update(obj, "confirmed") end for
66:
67: /* 2.7. CEGAR-NotEqual*/
68: notExplainedObjs = []
69: for obj in ListOfAccObjs do
70:     if obj.status != "unknown" and obj.explanation == "" then notExplainedObjs.add(obj) end
    if
71: end for
72: annotatedSC = attachAssertion(SC, notExplainedObjs, assertForm[0])
73: (confirmedObjs, explanations) = CEGAR.analyze(annotatedSC)
74: for i in [0, len(confirmedObjs)) do
75:     ListOfAccObjs.update(confirmedObjs[i], "confirmed", explanations[i])
76: end for
77: return ListOfAccObjs
```

This section explains the procedure for detecting register-access objects in source code (i.e., the second step in Algorithm 3) by combining PM, AI, BMC, and CEGAR. Algorithm 5 shows this process. The input of the process is SC - the

```
1  *(unsigned long*)var = 0x00000000;
2  /*@ assert not_access_BKP_CR: (unsigned long*)var != (unsigned long*)(0x40006C30)
       ;*/
```

(a) *NotEqual* assertion for Eva plugin

```
1   *(unsigned long*)var = 0x00000000;
2   /*@ assert access_BKP_CR: (unsigned long*)var == (unsigned long*)(0x40006C30);*/
```

(b) *Equal* assertion for Eva plugin

Figure 4.4: Examples of assertions for extracting actual register-access objects

target source code, `PatternRegrAcc` - the code patterns for register-access which are shown in Figure 4.4.1, `ListOfRegs` - list of registers in the target hardware where each register is described using a name, a valid accessed size, and an address, `ListOfLoopObjs` - information of loops in the target source code. The output is `ListOfAccObj` - a list of the register-access objects which are defined formally in Figure 4.3.3. However, at this step only *id*, *accessType*, *register*, *syntacticInfo*, *status*, *explanation* are extracted; *accessDetail*, *relations* will be extracted latter as explained in Section 4.4.1.

The process of detecting register-access objects includes two main steps. Lines from `5` to `9` show the first step. This step is to detect potential register-access expressions. In this step, expressions in the target source code are scanned using the `PatternRegisterAccess`. The matched expressions are considered potential register-access expressions.

Lines from `11` to `75` show the second step. This step is to detect actual register-access expressions among the potential ones. The general idea is to attach assertions after the positions of the potential expressions and execute these assertions to question whether these expressions are actual register-access expressions.

There are two formats of assertions declared at line `4`. These two formats are: *NotEqual* - assert whether an expression does not access the address of a register, *Equal* - assert whether an expression access the address of a register. Examples of assertions in these two formats can be found in Figure 4.4. The two formats of assertions bring different meanings if they are executed by different approximation approaches. If a *NotEqual* assertion is executed by an over-approximated approach (e.g., AI), the valid result means the expression is a `confirmedNot` register-access (i.e., $K_{over}, s0 \nvDash \mathbf{EF}\ p \iff$ `confirmedNot`); the invalid result means the expression is a `maybe` register-access (i.e., $K_{over}, s0 \models \mathbf{EF}\ p \iff$ `maybe`). However, if this assertion is executed by an under-approximation approach (e.g., BMC), the valid result has no meaning; the invalid result means the expression is a `confirmed` register-access (i.e., $K_{under}, s0 \models \mathbf{EF}\ p \iff$ `confirmed`). If an *Equal* assertion is executed by an over-approximated approach

(e.g., AI), the valid result means the expression is a `confirmed` register-access (i.e., $K_{over}, s0 \models \mathbf{AF}\ p \iff$ `confirmed`); the invalid result has no meaning. However, if the assertion is executed by the under-approximation approach (e.g., BMC), the valid result means the expression is a `confirmed` register-access (i.e., $K_{under}, s0 \models \mathbf{AF}\ p \implies K_{under}, s0 \models \mathbf{EF}\ p \iff$ `confirmed`); the invalid result has no meaning. In this algorithm, the *NotEqual* assertion format is applied before the *Equal* format because the *NotEqual* format can be analyzed by AI to exclude `confirmedNot` register-access objects. By excluding these objects, the computational burden is reduced for the latter steps.

Based on the two formats of assertions, we detect and explain actual register-access objects using AI, BMC, and CEGAR sequentially. The principle is that unsolved assertions by the former tools are left to the latter ones. As executing BMC takes more time and memory than AI, and executing CEGAR takes a greater amount of time than BMC, we apply the heavier ones later to reduce the amount of computation.

There are seven sub-steps in the second step. Steps 2.1 to 2.4 are to apply AI with different settings. In steps 2.1 and 2.2, executing *NotEqual* assertions by AI can detect `confirmedNot` register-access in cases the results of the assertions are valid or `maybe` register-access in cases the results of the assertions are invalid. In steps 2.3 and 2.4, executing *Equal* assertions by AI can confirm the `maybe` register-access detected by steps 2.1 and 2.2 in cases the results of the assertions are valid. `Unknown` assertions of AI are then examined by BMC in steps 2.5 and 2.6. BMC can detect `confirmed` register-access objects in cases the results of the *NotEqual* assertions are invalid or the results of the *Equal* assertions are valid. The remaining cases are marked as `unknown`. Counterexamples are generated if the status of the *NotEqual* assertions are invalid. These counterexamples are used as explanations for the register-access objects. Similarly, *NotEqual* assertions are employed in step 2.7 which applies CEGAR to explain the remaining register-access objects that BMC has not explained. For steps 2.2 and 2.4, only several loops, which directly affect a target expression, are unrolled. If the number of iterations for a loop is available in the `ListOfLoopObjs`, this number is used. Otherwise, the biggest number of iterations calculated in the `ListOfLoopObjs` is used. In this case, we only count on valid assertions if these loops are completely unrolled. Because these valid assertions have no meaning if the loops are partially unrolled. For steps 2.5 and 2.6, all loops are unrolled as is required by BMC.

We assume that loops in the same source code may share several properties. Hence, the biggest number of iterations calculated in the source code is applied for unrolling loops with `unknown` number of iterations.

Chapter 5

Meta-modeling microcontroller-based systems

5.1 Approach overview

In Chapter 3 and Chapter 4, we explained how to analyze and extract knowledge from the hardware manual and source code. Verifying the target coding rules usually requires the combination of multiple sources of information from both hardware and software. Since there are multiple sources of knowledge and the knowledge can be extracted by different techniques/ tools, there is a need to manage the knowledge to fulfill the verification target.

In this chapter, we explain how to apply model-driven engineering (MDE) techniques to represent the knowledge and verify the coding rules related to register-access. Specifically, we propose a meta-model in unified modeling language (UML) [34] to describe the structure of the three types of knowledge. Knowledge models of microcontroller-based systems are generated by instantiating the meta-model using the extracted knowledge from the source code and hardware manual. We use the Eclipse modeling framework (EMF), and the Ecore modeling tool [37] to design the meta-model and generate the Java code for instantiating these knowledge models. The knowledge models can be questioned systematically to verify the target coding rules.

Using the knowledge models for verifying the target coding rules has several advantages. The first advantage is that the power of multiple techniques can be systematically gathered. The second advantage is that the knowledge can be reused to check multiple coding rules. The third advantage is that once the knowledge models are available, the coding rule can be interactively checked. Details of the knowledge meta-model can be found in Section 5.2.

Figure 5.1 shows the overview of the approach for modeling the knowledge.

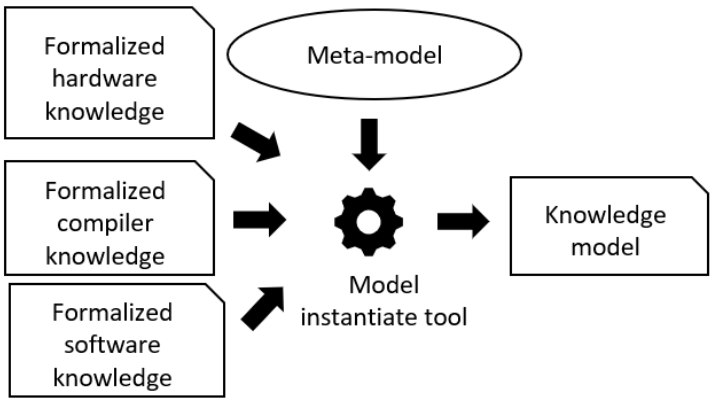

Figure 5.1: Overview of the process for modeling knowledge

The input of the modeling process is the knowledge to be modeled (i.e., *Formalized hardware knowledge*, *Formalized compiler knowledge*, *Formalized software knowledge*) and a meta-model to present the structure of the input knowledge. The output of the modeling process is a knowledge model. Among the three types of knowledge, *Formalized hardware knowledge* is described in Chapter 3, and *Formalized software knowledge* is described in Section 4.3. Here, we define the *Formalized compiler knowledge*.

In a compiler, data type, size, and range information will decide the presentation of registers in a C program. In C language, sizes of registers are represented by data types (e.g., `(unsigned long*) 0xFFF20338` is the representation of a 32-bit register in some compiler). For different compilers, the sizes of data types may be different. Hence, it is necessary to provide this information in advance. We define the compiler knowledge as follows.

Definition 5.1.1 (Compiler knowledge) *A compiler knowledge CK is a tuple CK* $=$ ***<id, name, pairOfTypeAndSize>****, where*

id $\in \mathbb{N}$*;*
name $\in$ ***S****;*
pairOfTypeAndSize $=$ ***[pairOfTypeAndSize_0, .., pairOfTypeAndSize_n];***
n $\in \mathbb{N}$
pairOfTypeAndSize_i $=$ ***<dataType, size, lowerBound, upperBound>***
dataType $\in$ ***S****;* ***S*** *is a set of strings,*
size $\in \mathbb{N}*$, ***lowerBound*** *is* $\mathbb{N}$, ***upperBound*** *is* $\mathbb{N}$

For example, the formalization of a compiler named **sampleCompiler** with three datatypes used to represent register, `unsigned long` - **32** bits, `unsigned short` - **16** bits, and `unsigned char` - **8** bits, is as follows.

<0, **sampleCompiler**, [<unsigned long, 32, 0, 18446744073709551615>,
<unsigned short, 16, 0, 65535>, <unsigned char, 8, 0, 255>]>

The information about the compiler can be obtained by checking the compiler manual. In this research, we assume that the knowledge of compilers is given in the format shown in the above definition.

5.2 Knowledge meta-model

Figure 5.2 shows the proposed meta-model in UML. This meta-model is designed to represent the hardware, compiler, and software knowledge for verifying the target coding rules. There are four packages in this meta-model: *HWAndCompiler* - hardware and compiler package, *SWSyntactic* - software syntactic, *SWSemantic* - software semantic, and *Supplementary*. The textitHWAndCompiler package is for representing the knowledge of employed hardware and compiler. Details of this package are shown in Section 5.2.1. The *SWSyntactic* represents the syntactic knowledge of the source code. Details of *SWSyntactic* package are shown in Section 5.2.3. On the other hand, the *SWSemantic* package represents the semantic knowledge of the source code. Details of *SWSemantic* package are shown in Section 5.2.2. Finally, the *Supplementary* package is for additional necessary information for the verification process. Details of *Supplementary* package are shown in Section 5.2.4.

5.2.1 Hardware and compiler package

Hardware and compiler knowledge is represented in *HWAndCompiler* package. Figure 5.3 shows this package. There are six classes in this package: *Configuration*, *HardwareKnowledge*, *CompilerKnowledge* , *PairTypeAndSize*, *Register*, and *CodingRule*. The root class in this package is *Configuration*. A *Configuration* object is the composition of a *HardwareKnowledge* object and a *CompilerKnowledge* object. As explained in Chapter 3, the hardware knowledge includes registers which are represented by class *Register*, and coding rules in using these registers, which are represented by class *CodingRule*. The compiler knowledge includes information about data types' sizes represented by class *PairTypeAndSize*.

Figure 5.4 shows a representation of a microcontroller named *microcontroller1* with two registers: *register1* at the address *0xFFF20300* and *register2* at the address *0x40006C30*. There is a coding rule for using these two registers: *After setting register1 to 0x00000010, do not set register2[1:0] to 00.* The compiler in this example is named *sampleCompiler* There are three pairs of types and sizes: *unsigned long* with the size 32, the lower bound 0, the upper bound

18446744073709551615; *unsigned short* with the size is 16, the lower bound is 0, the upper bound is 65535; *unsigned char* with the size is 8, the lower bound is 0, the upper bound is 255.

5.2.2 Software semantic package

Figure 5.5 shows this the *Software semantic package*. As knowledge related to register-access is the main focus of this package, an abstract class named *Register-Access* is the root class in this package. Two concrete classes extend this abstract class: *WriteAccess* and *ReadAccess*. Both a *WriteAccess* and a *ReadAccess* objects have an *id*, an access size represented by a data type in C, an accessed register which is represented by a *Register* object, and an expression which perform the access which is represented by an *Expression* object, an access status which is represented by an *AccessStatus* object. In addition, a *WriteAccess* object has the possible written values, which is represented by a *WrittenValueSet* while a *ReadAccess* object has the possible accessed bits, which is represented by an *AccessBitSet* object.

There are two special data types for presenting semantic knowledge besides basic data types such as string and integer. The first data type is a boolean value with a precision level. Abstract class *BooleanValue* is used to represent boolean data. This class has an attribute named *value* for the boolean value itself and an attribute named *preciseness* to represent this value's reliability level. Two concrete classes extend this abstract: *AccessStatus* and *OrderStatus*. Class *AccessStatus* represents whether a register is accessed. Class *OrderStatus* represents execution orders among register-accesses.

The second data type is non-negative integer sets. Data, including accessed bits of a read-access and the written value of a write-access can be represented as a non-negative integer. Abstract class *IntegerSet* is used to represent integer set data. Three concrete classes are extending from this abstract class: *Enumeration*, *Interval*, and *PeriodicityInterval*. As the integer set data is used to represent the written values and accessed bits, an *WrittenValueSet* and an *AccessBitSet* object are composited by either a *Enumeration*, an *Interval*, or a *PeriodicityInterval* object.

Figure 5.6 shows a representation of the sample source code in Figure 5.7 using the *Software semantic package*. There are two write-access objects at lines 6 and 9. These two register-access objects are represented in objects named *wa1* and *wa2* in Figure 5.6. Assuming that the register-access objects are detected by an over-approximation approach like AI, the status of these two objects is *MAYBE* which is shown in objects named *aStatus1* and *aStatus2*. The written value of *wa1* is `0x00000010`. Assume that this value is not a special value, this value is represented by an object named *n1*. The written value of *wa2* is `0x00000000`. Assume that this value is a *CONDITIONAL_INVALID* value, this value is represented by an object

named *n2*. The relation of the two write-access objects is represented by objects named *exeOrder1* and *exeOrder2*. Logically, one order status object is sufficient to show the relation between the two register-access objects. We store two objects for order status because it will be faster to query the model. Concretely, if we want to check write-access objects which occur after *wa1*, checking *exeOrder1* is the optimal way. On the other hand, if we want to check write-access objects which occur after *wa2*, checking *exeOrder2* is the best solution.

5.2.3 Software syntactic package

Figure 5.8 shows this the *Software syntactic package*. A software syntactic knowledge package divides a source code into file, function, and line levels. In correspondingly, we have four classes: *SourceCode*, *File*, *Function*, *Line*. Class *SourceCode* has one attribute, which is the name. Class *File* has two attributes: fileName and filePath. Class *Function* has two attributes: functionName and scope. There are two possible values for the scope attribute: local - visible for the file in which the function is defined only, and public - visible for other files. Class *Expression* represents the C expressions that match code patterns for register-access. Expressions that match the code patterns are called potential register-access expressions. We employ the code patterns described in 4.4.2 for distinguishing the register-access expressions from other C expressions. Each *Expression* object has a unique id, a matched pattern's name, which is either *RA* for read-access or *WA* for write-access and the expression in string format.

Figure 5.9 shows the representation of the syntactic feature of the source code in Figure 5.7. Object *sc* represents the source code with its name. Object *f1* represents the file named *main.c*. Object *func1* represents the function *main*. Objects *line1* to *line9* represent the code lines in the source code. Objects *exp1* and *exp2* represent the two expressions that match the code patterns for register-access.

5.2.4 Supplementary package

Figure 5.10 shows this the *Supplementary package*. The supplementary package represents an explanation for the extracted software knowledge. There are three supplementary pieces of information: the syntactic feature of an expression to show the expression matches a code pattern shown by class *ExpressionComponent*, the execution trace of a boolean knowledge is shown by class *TraceInfo*, and the name of the method/tool used to extract the knowledge is shown by class *ExtractionMethod*.

Figure 5.11 shows instances of class *ExpressionComponent*. Object *wAComp1* and *wAComp2* show the expression components in the two expressions at lines *6* and *9* in Figure 5.7. Figure 5.12 shows trace information from the program's starting point to the execution point of the expression at line *6*. Figure 5.13 shows

an example of extraction methods. Object *extractMethod1* represents the extraction method using the Eva plugin with no configuration. Object *extractMethod2* represents the extraction method using the CBMC with a configuration that the maximum depth to be explored is *1000*.

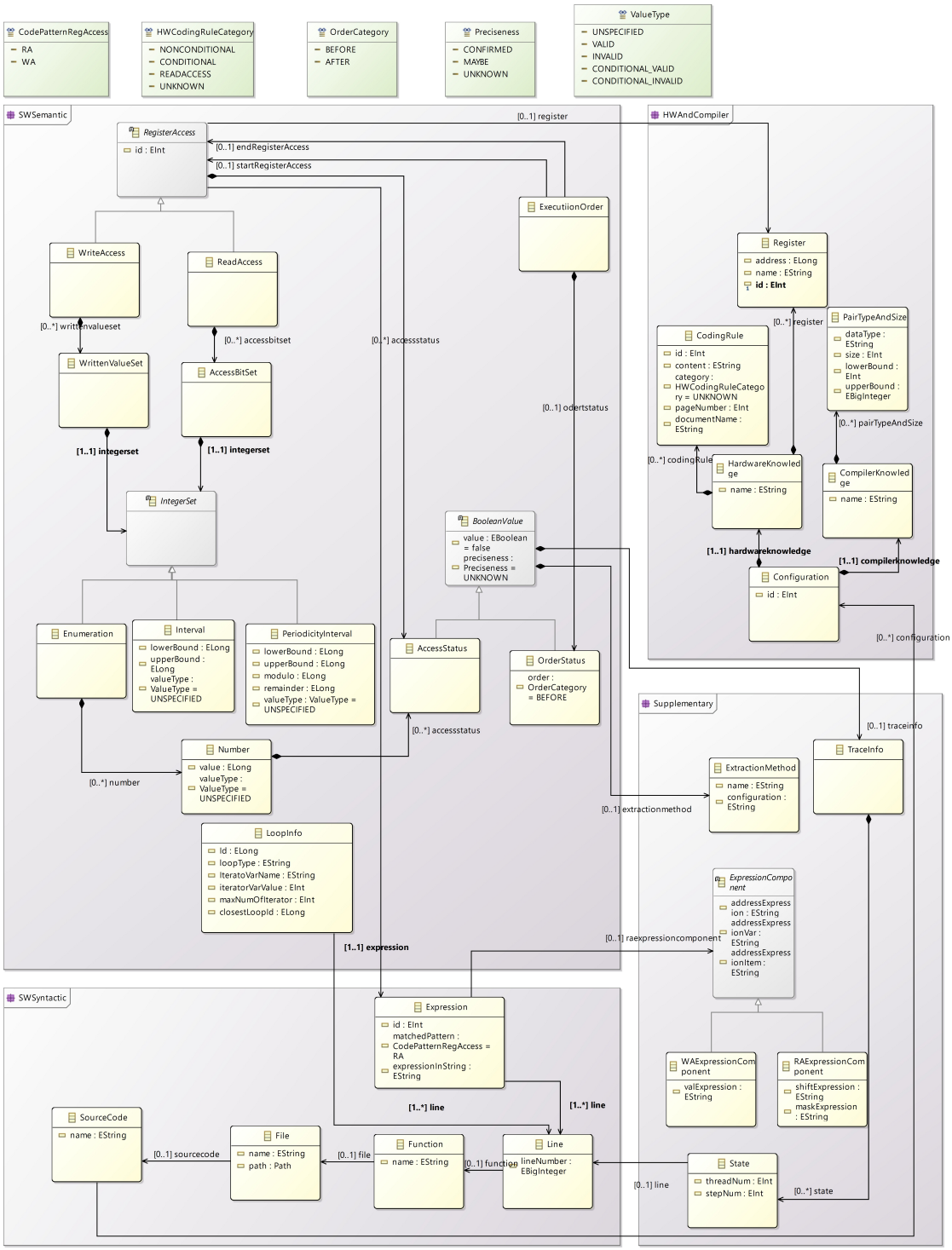

Figure 5.2: Knowledge meta-model

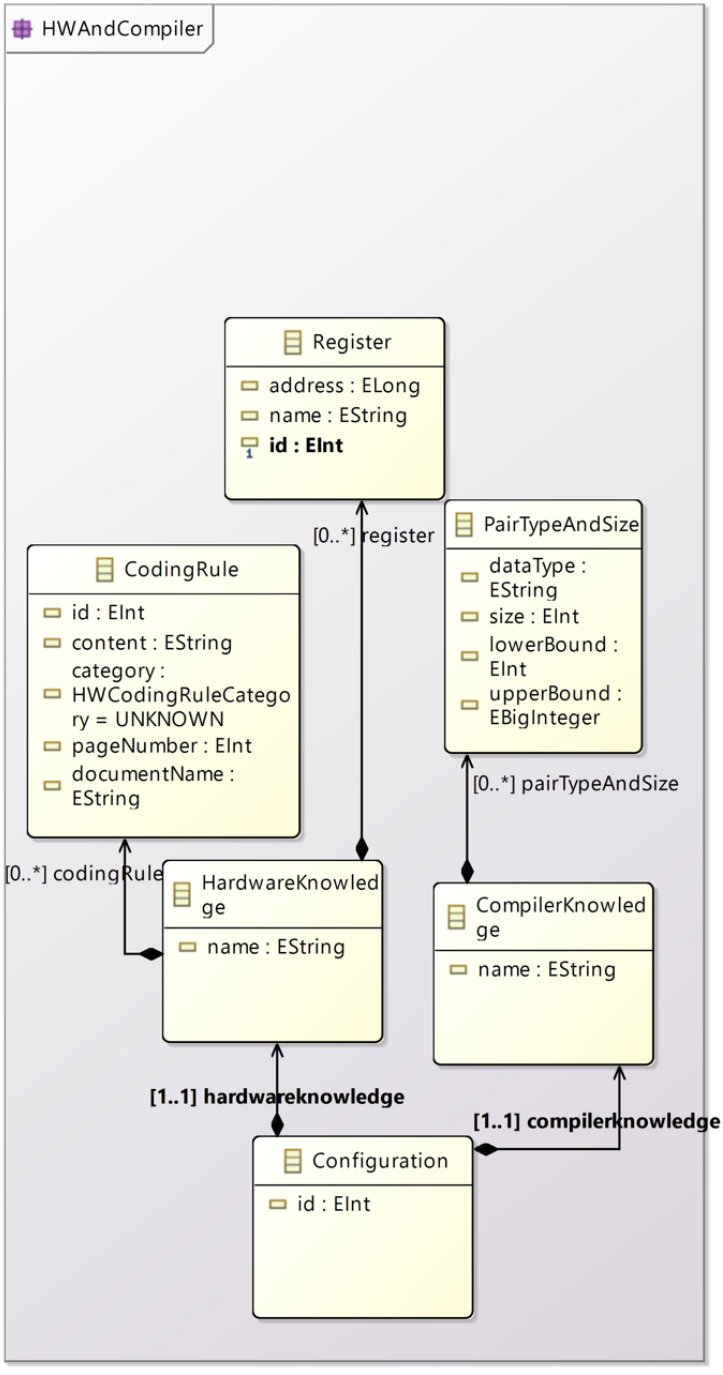

Figure 5.3: Hardware and compiler package

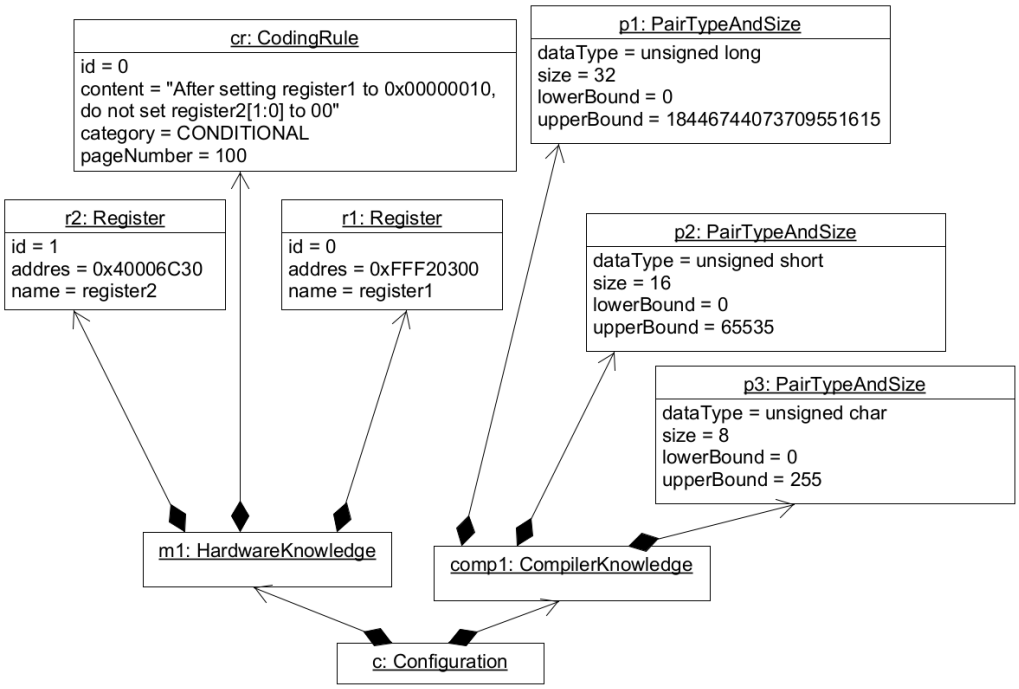

Figure 5.4: Example of hardware and compiler package

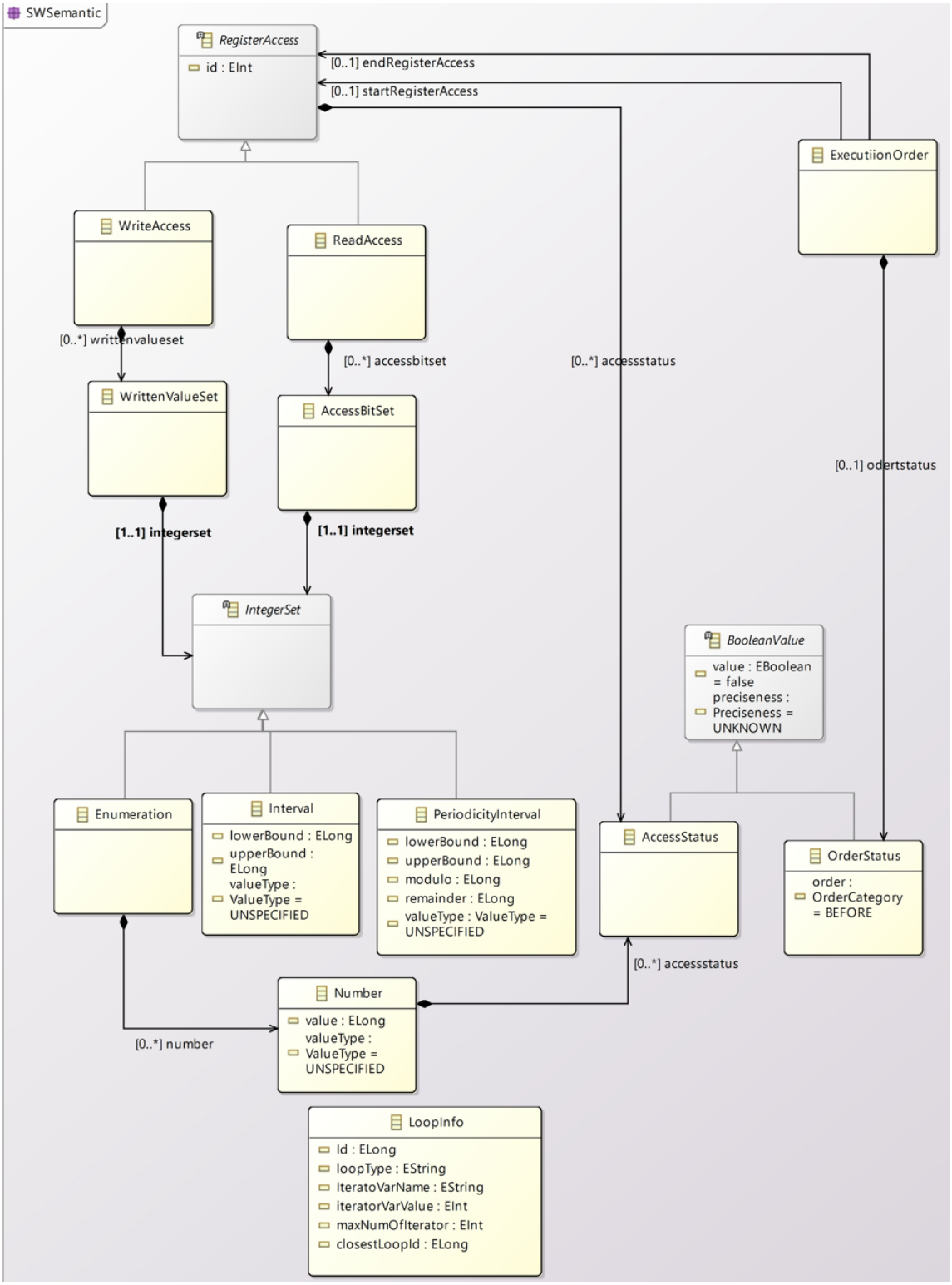

Figure 5.5: Software semantic package

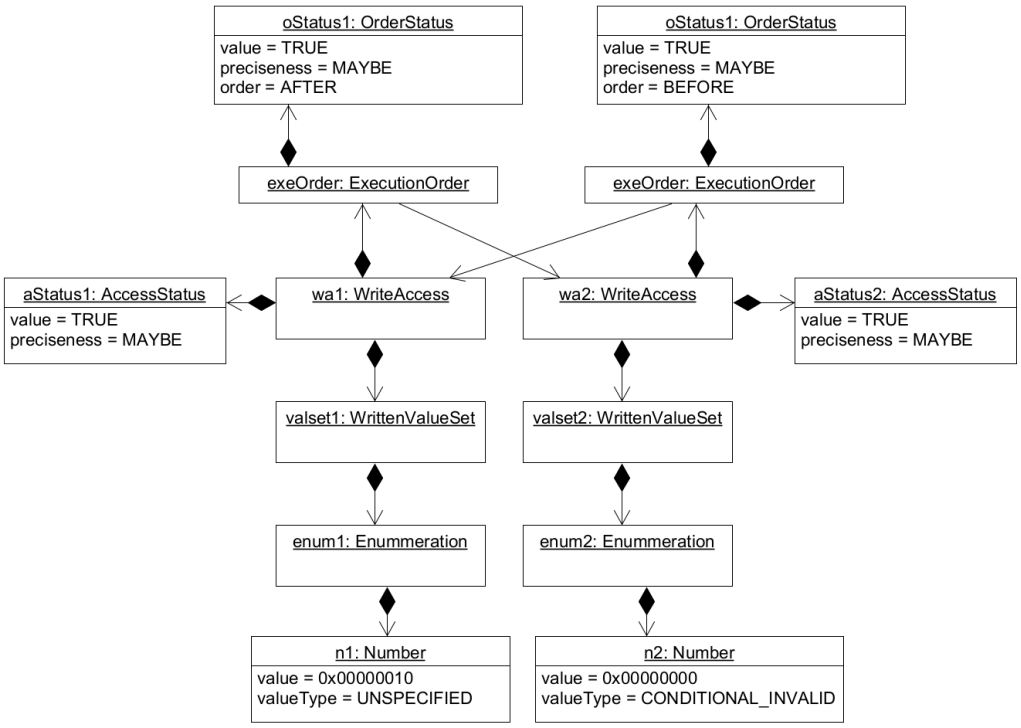

Figure 5.6: Example of software semantic package

```
1  void main(void) {
2      unsigned long* register_1 = (unsigned long*) 0xFFF20300;
3      unsigned long* register_2 = (unsigned long*) 0x40006C30;
4
5      int n = 1000;
6      *register_1 = 0x00000010;
7
8      if (n == 1000) {
9          *register_2 =  0x00000000;
10     }
11
12 }
```

Figure 5.7: Sample source code

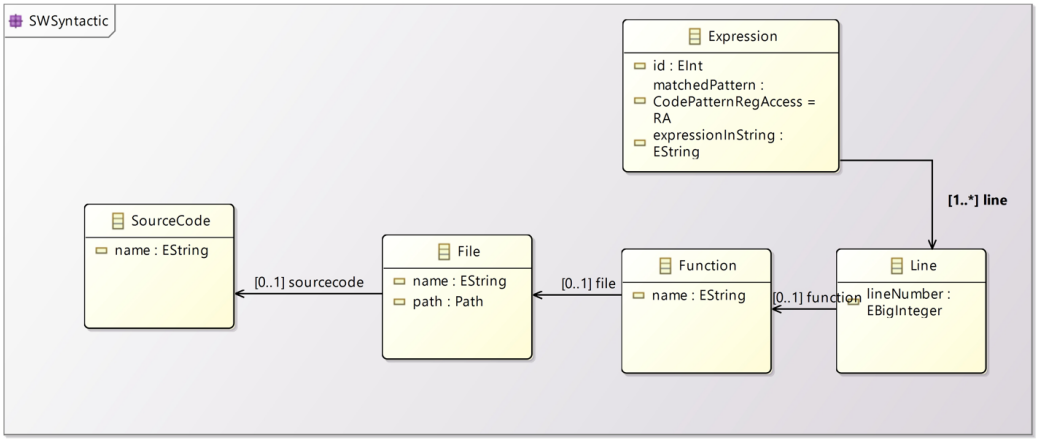

Figure 5.8: Software syntactic package

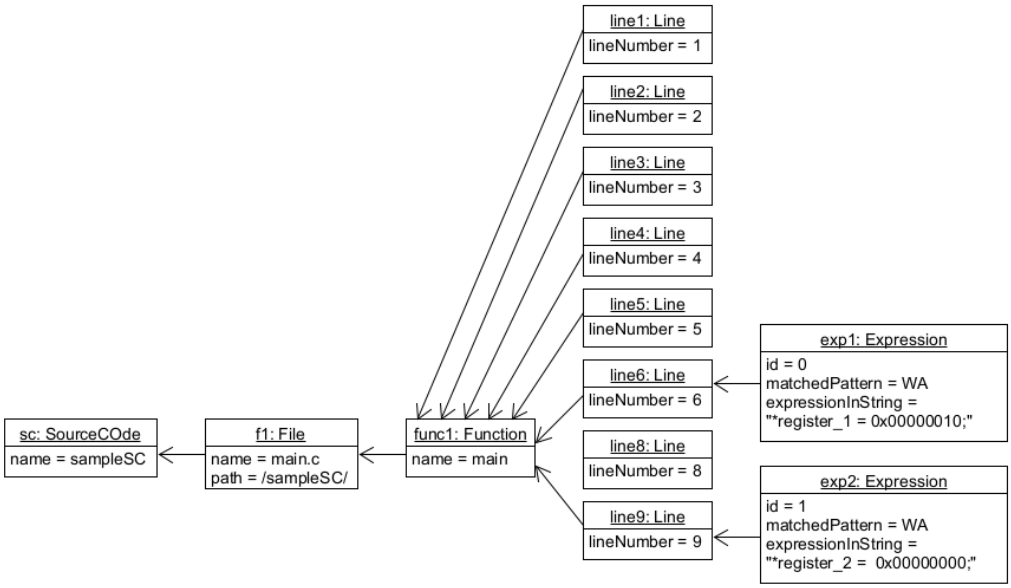

Figure 5.9: Example of software syntactic package

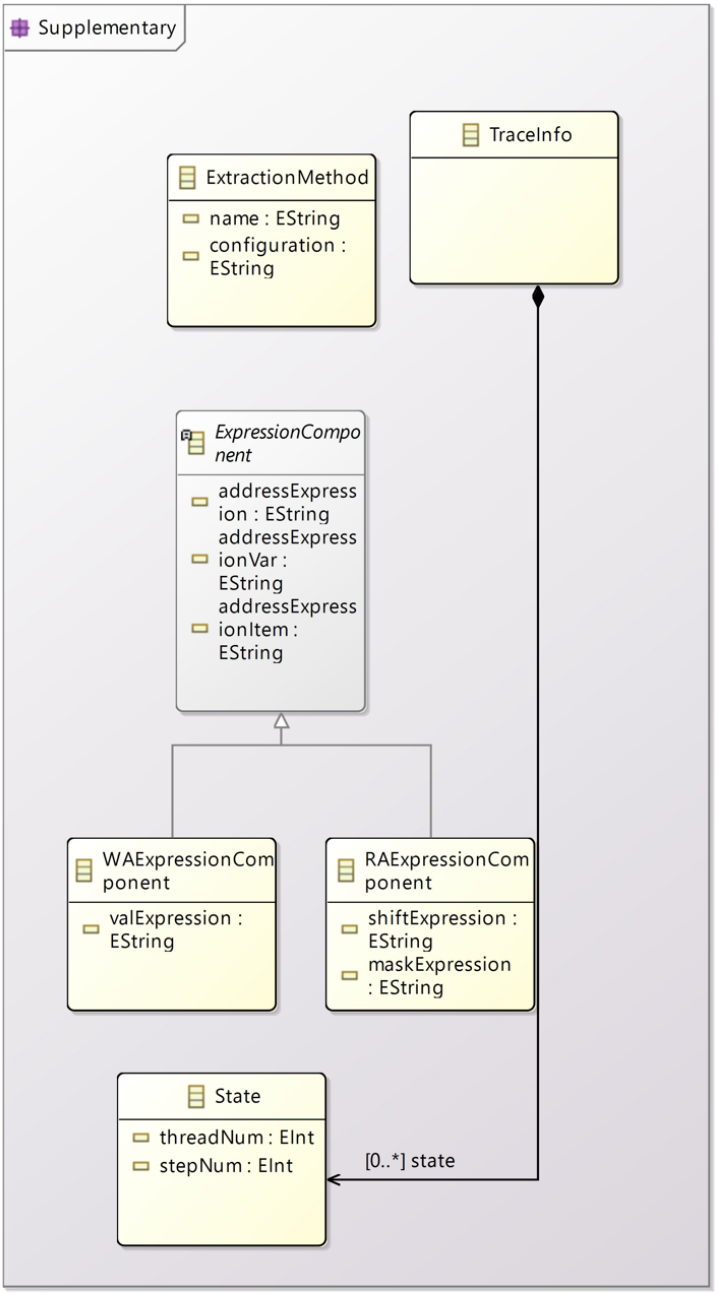

Figure 5.10: Supplementary package

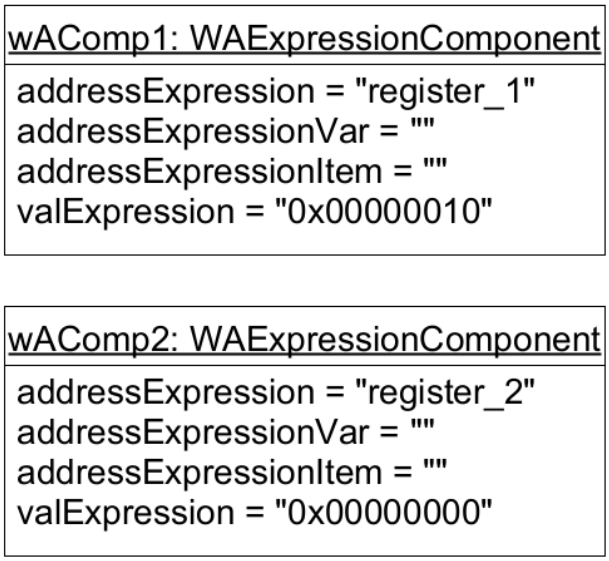

Figure 5.11: Example of expression component

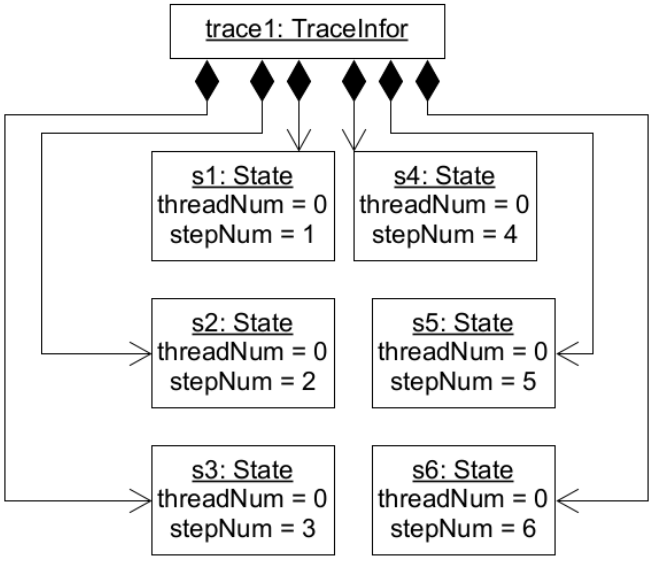

Figure 5.12: Example of expression component

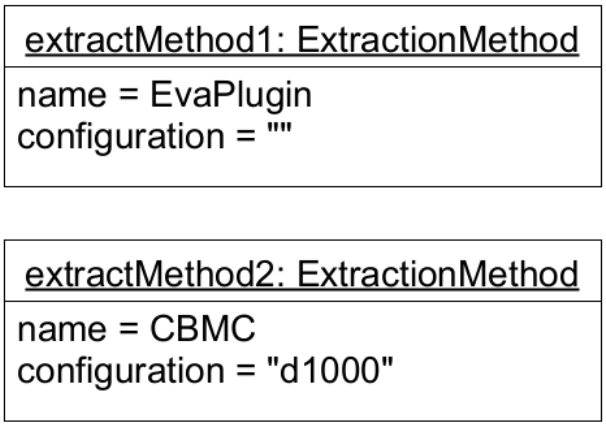

Figure 5.13: Example of extraction methods

Chapter 6

Verifying systems against microcontroller-specific coding rules

6.1 Approach overview

The previous chapter discussed how hardware, compiler, and software knowledge is gathered as a knowledge model. This chapter explains how to use this model to verify the system against the target coding rules. As the knowledge model is described in UML, we naturally think of query techniques for the UML model as the solution for the verification task.

Figure 6.1 shows an overview of the process for verifying the system using the knowledge model. The input of this process is a model - an instance of the meta-model in the previous chapter, a coding rule in the form of a query, and a list of pre-defined queries. The output is a *verification result* which is defined below.

Definition 6.1.1 (Verification result) *A verification result VR = {<status, regAccObjs>}, where*
status ∈ {notViolated, violated, maybeViolated, unknown};
regAccObjs ∈ {read-access-object, write-access-object, <condObjs, reqObj>};
condObjs is a list of write-access objects;
reqObj is a write-access object.

A *verification result* is a list of double tuples of a *status* and a *registerAccessObjects*. A status of a *verification result* can be *notViolated*, *violated*, *maybeviolated*, or *unknown*. This status is decided differently based on the coding rule categories and constraint type. Table 6.1 shows the factors to decide the status. For constraints related to accessibility or access size of *read-access* coding rules and *non-conditional-case* coding rules (i.e., read-access-constraint, NotWriteable,

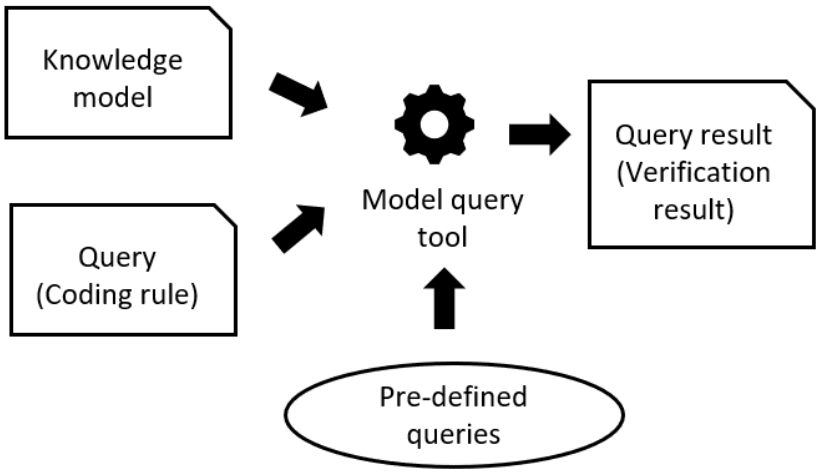

Figure 6.1: Verification approach

NotWriteableBit, WriteValidSize, and WriteInvalidSize), we need to consider the precision level of the register-access object only. For constraints related to written values of *non-conditional-case* coding rules (i.e., ValidValue, InvalidValue, ValidValueBit, and InvalidValueBit), we need to consider the precision level of the register-access object and the precision level of the written value. For *condition-case* coding rules, we need to consider the precision level of the register-access objects, the precision level of the written values, and the precision level of the temporal relationships among the condition and the requirement register-access objects. There are three levels of precision as defined in Definition 4.3.3: *confirmed*, *maybe*, and *unknown*. A status of a *verification result* is *notViolated* if there are no register-access objects which violate the coding rule. A status of a *verification result* is *violated* if all precision level is *confirmed*. A status of a *verification result* is *maybeViolated* if there is a level is *maybe*, and no level is *unknown*. A status of a *verification result* is *unknown* for the remaining cases in which there is an *unknown* level.

If the status of checking a coding rule is *notViolated*, there are possible undetected bugs if the bugs are out of our current scope which are the expressions that match our code pattern for register-access and C single thread C program with no interruptions. If the verification result of the coding is *maybeViolated* or *unknown*, there are possible false warnings. If the verification result of the coding is *violated*, we can confirm that this is an actual violation.

The process of querying the knowledge model is a model-to-model transformation. The input and output models are instances of the meta-model described in Figure 5.2. Each model is a set of objects where the output model is a subset of the input. The applied query is a filter to get violated objects in the input model.

Table 6.1: Factors to decide the statuses of a *verification result*

Coding rule categories	Constraint type	Factors to consider
read-access non-conditional-case	read-access-constraint NotWriteable NotWriteableBit WriteValidSize WriteInvalidSize	Precision of access status
	ValidValue InvalidValue ValidValueBit InvalidValueBit	Precision of access status Precision of written value
conditional-case	ValidValue InvalidValue ValidValueBit InvalidValueBit	Precision of access status Precision of written values Precision of temporal relation

The main difficulty of verifying the coding rules is the large number of variations of coding rules. As not all developers are familiar with the query language, we provide predefined queries for the defined categories of coding rules to facilitate the usage of the verification approach. The pre-defined queries will be explained in Section 6.2.

6.2 Pre-defined queries for target coding rules

As discussed in Section 2.6.3, this research uses QVTo language for describing queries. Like many popular high-level programming languages such as C or C++, QVTo is imperative. However, not all developers are familiar with QVTo. The requirement of learning a new language may prevent developers from applying the proposed verification process to their daily work. However, as we discussed in Chapter 3, the target coding can be categorized into several groups based on the structure of the coding rules. Coding rules in the same category share similar queries. Hence, there is room for providing support in writing queries. The solution is that we provide pre-defined queries for coding rules in the known groups. This section explains the queries in detail.

In Chapter 3, we categorized the coding rules based on their structures (i.e., *read-access*, *non-conditional-case*, and *conditional-case*). For the *read-access* category, there are three kinds of requirements: the readability of a whole register, the readability of a bit of a register, and the access sizes of a register. The

requirement can be expressed as a list of valid access sizes or a list of invalid access sizes for access sizes. The query for *read-access* coding rules on the readability of whole registers is explained in Section 6.2.1. The query for checking the *register-access* coding rules on the valid access sizes is shown in Section 6.2.2

For the *non-conditional-case* coding rule in the *write-access* category, the requirement is used to describe the requirements on how a write-access is performed on a register. For the *write-access*, there are five kinds of requirements: on the writability of a whole register, the writability of a bit of a register, on the access sizes of a register, on the written values of a whole register, on the written values of a bit of a register. For the written value (or the access size of a register), the requirement can express a list of valid values or a list of invalid values (or a list of valid access sizes or a list of invalid access sizes). The query on checking the *non-conditional-case* coding rules on the valid written values is discussed in Section 6.2.3. The query on checking the *non-conditional-case* coding rules on the valid written values is discussed in Section 6.2.4.

For *conditional-case* coding rules, there are two parts: the condition and the requirement. The condition and requirement parts can be represented in the same formats as the *non-conditional-case* coding rules. However, for the *conditional-case* coding rule, one must consider the temporal properties to represent the relationship between the condition and requirement parts which is either *before*, *after*, *before-or-after*. The query for the *conditional-case* coding rules with the single condition and *before* as the temporal property is discussed in Section 6.2.5. The query for the *conditional-case* coding rules with the single condition and *after* as the temporal property is shown in Section 6.2.6. The query for the *conditional-case* coding rules with the single condition and *before-or-after* as the temporal property is shown in Section 6.2.7.

6.2.1 Read-access query on the readability

Algorithm 6 Query for read-access coding rules - Not readable register

Input: `knowledgeModel`, `regName`,
Output: `violatedObjList`
Procedure:
violatedObjList = []
for obj in knowledgeModel **do**
  if obj is ReadAccess and obj.register.name == regName and (obj.preciseness.toString() == "CONFIRMED" or obj.preciseness.toString() == "MAYBE") **then**
    violatedObjList.add(obj)
  end if
end for
return violatedObjList

Algorithm 6 shows the query for read-access coding rules regarding readability. The input of this algorithm is a knowledge model and a registered name in a string. This query is to check whether there are any register-access objects in the knowledge model that performs the read-access operator over the input register. The output is a list of register-access objects if these objects read the input register, and an empty list otherwise. The procedure for performing this query is to iterate over each read-access in the knowledge model, then check whether they access the target register to find the violated ones.

6.2.2 Register-access query on the valid access sizes

Algorithm 7 Query for read-access coding rules - Valid read-access size

Input: `knowledgeModel`, `regName`, `listValidSize`
Output: `violatedObjList`
Procedure:
violatedObjList = []
for obj in knowledgeModel **do**
if obj is ReadAccess and obj.register.name == regName and (obj.preciseness.toString() == "CONFIRMED" or obj.preciseness.toString() == "MAYBE") **then**
if obj.pairOfTypeAndSize.size not in listValidSize **then**
violatedObjList.add(obj)
end if
end if
end for
return violatedObjList

Algorithm 7 shows the query for checking the access size of read-access. The input is a knowledge model, a register name in string format, and a list of valid sizes. This query is to check whether there is any register-access over the target register which has an access size outside of the valid list. The output is a list of register-access objects if these objects read the input register with an invalid size, and an empty list otherwise. The procedure for performing this query is to iterate over each read-access in the knowledge model, then check whether they access the target register with an invalid size to find the violated ones.

6.2.3 Non-conditional-case query on the valid written values

Algorithm 8 Query for non-conditional-case coding rules - Valid written values

```
Input: knowledgeModel, regName, listOfValidVal
Output: violatedObjList
Procedure:
violatedObjList = []
for obj in knowledgeModel do
  if obj is WriteAccess and obj.register.name == regName and (obj.preciseness
== "CONFIRMED" or obj.preciseness == "MAYBE") then
    for valSet in obj.writtenvalueset do
      if valSet isTypeOf Enumeration then
        for val in valSet do
          if val not in listOfValidVal and (val.preciseness == "CONFIRMED"
or val.preciseness == "MAYBE") then
            violatedObjList.add(obj)
          end if
        end for
      end if
    end for
  end if
end for
return violatedObjList
```

Algorithm 8 is a query for non-conditional write-access regarding the valid written value. The input of this algorithm is a knowledge model, a register name, and a list of valid values in form of an integer list. This query is to check whether there are any register-access objects in the knowledge model that performs the write-access operator over the input register with a value outside of the valid list. The output is a list of register-access objects if these objects write to the input register. The procedure for performing this query is to iterate over each write-access in the knowledge model, then check whether they access the target register with an invalid value to find the violated one.

6.2.4 Non-conditional-case query on the invalid written values

Algorithm 9 Query for non-conditional-case coding rules - Invalid written values

```
Input: knowledgeModel, regName, listOfInvalidVal
Output: violatedObjList
Procedure:
violatedObjList = []
for obj in knowledgeModel do
  if obj is WriteAccess and obj.register.name == regName and (obj.preciseness
== "CONFIRMED" or obj.preciseness == "MAYBE") then
    for valSet in obj.writtenvalueset do
      if valSet isTypeOf Enumeration then
        for val in valSet do
          if val in listOfInvalidVal and (val.preciseness == "CONFIRMED"
or val.preciseness == "MAYBE") then
            violatedObjList.add(obj)
          end if
        end for
      end if
    end for
  end if
end for
return violatedObjList
```

Algorithm 9 is a query for non-conditional write-access regarding the invalid written value. The input of this algorithm is a knowledge model, a register name, and a list of invalid values in form of an integer list. This query is to check whether there are any register-access objects in the knowledge model that performs the write-access operator over the input register with a value inside of the invalid list. The output is a list of register-access objects if these objects write to the input register. The procedure for performing this query is to iterate over each write-access in the knowledge model, then check whether they access the target register with an invalid value to find the violated one.

6.2.5 Conditional-case query with single condition and *before* as the temporal property

Algorithm 10 Query for conditional-case coding rules - single condition - before

```
Input: knowledgeModel, condRegObj, reqRegObj
Output: violatedObjList
Procedure:
violatedObjList = []
for oi in knowledgeModel.RegisterAccessObjs do
  if oi satisfies condRegObj and (oi.preciseness == CONFIRMED or
oi.preciseness == MAYBE) then
    violatedFlag = true
    for oj in oi.getBeforeObj() do
      if oj satisfies reqRegObj and (oj.preciseness == CONFIRMED or
oj.preciseness == MAYBE) then
        violatedFlag = false
        break
      end if
    end for
    if violatedFlag == true then
      violatedObjList.add(oi)
    end if
  end if
end for
return violatedObjList
```

Algorithm 10 is a query for a conditional write-access with temporal property is *before*. The input of this algorithm is a knowledge model, a register-access object for representing the condition, and a register-access object for representing the requirement. This query is to check before the condition object holds in the knowledge model, whether there is an object which satisfies the requirement. The output is a list of register-access objects which satisfy the condition object but there is no object executed before that satisfies the requirement.

6.2.6 Conditional-case query with single condition and *after* as the temporal property

Algorithm 11 Query for conditional-case coding rules - single condition - after

```
Input: knowledgeModel, condRegObj, reqRegObj
Output: violatedObjList
Procedure:
violatedObjList = []
for oi in knowledgeModel.RegisterAccessObjs do
  if oi satisfies condRegObj and (oi.preciseness == CONFIRMED or
oi.preciseness == MAYBE) then
    violatedFlag = true
    for oj in oi.getAfterObj() do
      if oj satisfies reqRegObj and (oj.preciseness == CONFIRMED or
oj.preciseness == MAYBE) then
        violatedFlag = false
        break
      end if
    end for
    if violatedFlag == true then
      violatedObjList.add(oi)
    end if
  end if
end for
return violatedObjList
```

Algorithm 11 is a query for a conditional write-access with temporal property is *after*. The input of this algorithm is a knowledge model, a register-access object for representing the condition, and a register-access object for representing the requirement. This query is to check before the condition object holds in the knowledge model, whether there is an object which satisfies the requirement. The output is a list of register-access objects which satisfy the condition object but there is no object executed after that satisfies the requirement.

6.2.7 Conditional-case query with single condition and *before-or-after* as the temporal property

Algorithm 12 Query for conditional-case coding rules - single condition - before or after

Input: `knowledgeModel`, `condRegObj`, `reqRegObj`
Output: `violatedObjList`
Procedure:
for oi in knowledgeModel.RegisterAccessObjs **do**
  if oi satisfies condRegObj and (oi.preciseness = CONFIRMED or oi.preciseness = MAYBE) **then**
    violatedFlag = true
    for oj in knowledgeModel.RegisterAccessObjs **do**
      if oj satisfies reqRegObj and (oj.preciseness = CONFIRMED or oj.preciseness = MAYBE) **then**
        violatedFlag = false
        break
      end if
    end for
    if violatedFlag == true **then**
      violatedObjList.add(oi)
    end if
  end if
end for
return violatedObjList

Algorithm 12 is a query for a conditional write-access with temporal property is *before-or-after*. Actually, *before-or-after* means that there is no requirement on the executed order among register-access objects. The input of this algorithm is a knowledge model, a register-access object for representing the condition, and a register-access object for representing the requirement. This query is to check if the condition object holds in the knowledge model, and whether the requirement object holds. The output is a list of register-access objects which satisfies the condition object but there is no object that satisfies the requirement.

Chapter 7

Implementation architecture

This chapter first explains the implementation architecture of the verification framework. Details of this implementation can be found in Section 7.1. Subsequently, the verification process using the framework is described in Section 7.2.

7.1 Architecture overview

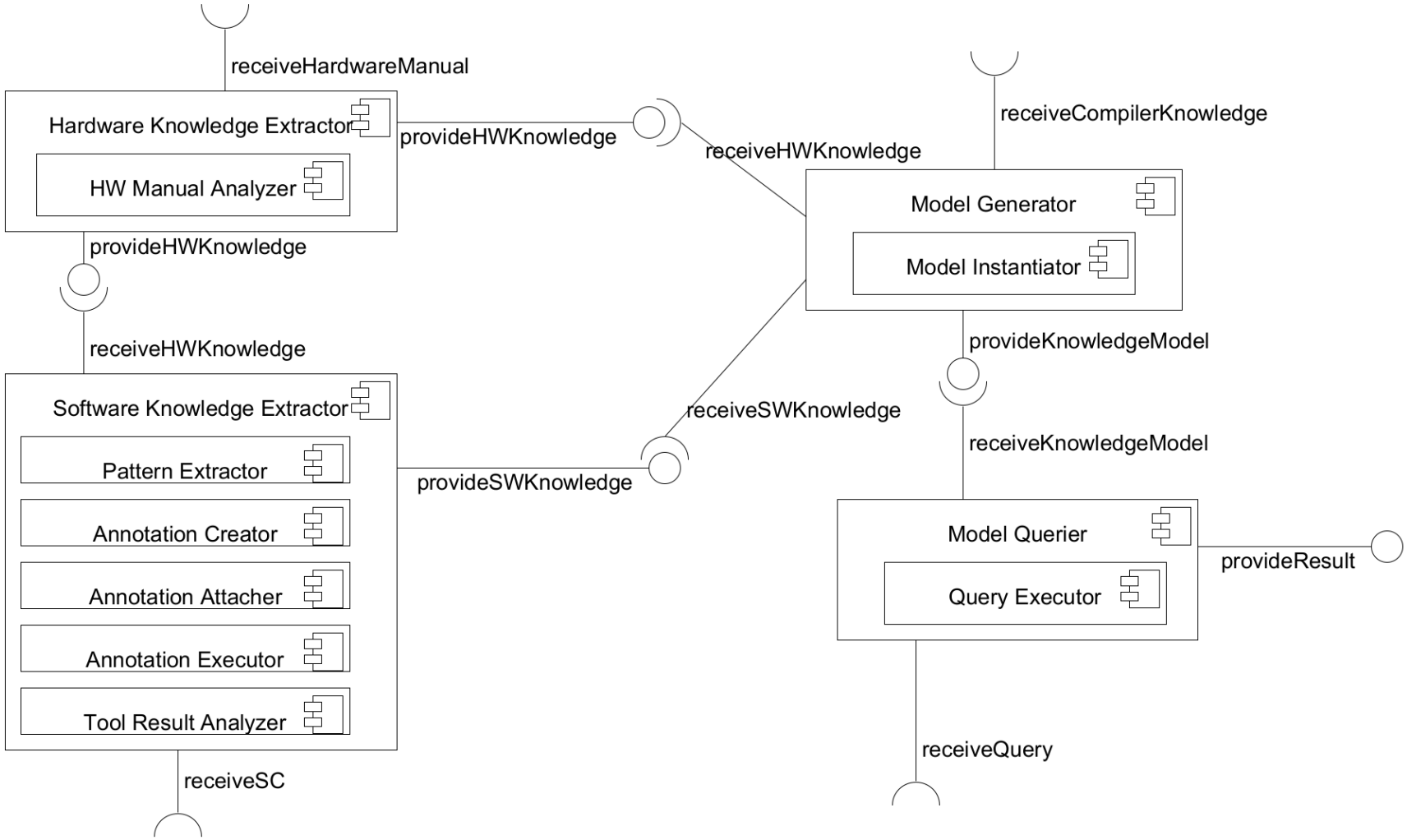

Figure 7.1: Implementation architecture

Figure 7.1 shows the overview of the implementation architecture for the verification system. The input of this framework includes a hardware manual in PDF, a C program that can be compiled by GCC, knowledge of a compiler in the format of Definition 5.1.1, and a query that can be executed by EMF. The output of this framework is the verification in the format of Definition 6.1.1.

The framework includes four main components: *Software Knowledge Extractor*, *Hardware Knowledge Extractor*, *Model Generator*, and *Model Querier*. This section will explain each component in detail.

Software Knowledge Extractor is a component implemented in python for extracting information related to register-access in hardware-dependent source code. This component can extract the following two kinds of information. The first information is related to the loop, including syntactic information (such as the line number and file name) and the maximum number of iterations. The second information relates to register-access, including syntactic information and semantic information. The semantic information includes an accessed register, accessed size, written value for writer-access, accessed bits for read-access, and execution order among these accesses.

The component includes 5 modules: *Pattern Extractor*, *Annotation Creator*, *Annotation Attacher*, *Anotation Executor*, and *Tool Result Analyzer*. *Pattern Extractor* is used to extract target code expressions using Cobra. This model detects read-access expressions which read the whole register value, write-access expressions, read-access expressions which read several bits in registers, and for and while loop in source code. *Annotation Creator* can create assertions to extract target knowledge. Created assertions are different among tools. *Annotation Attacher* is responsible for attaching the created assertion to the source code. *Annotation Executor* is used to execute the annotated source code. Finally, *Tool Result Analyzer* will extract target knowledge from the output of program analysis tools (Eva plugin, CBMC, SatAbs). This component is implemented using python.

Hardware Knowledge Extractor is a component implemented in python for extracting knowledge from the hardware manual. The input of this module is a hardware manual in PDF format. This module's output is the hardware knowledge in the format shown in 3.3. This component contains the *HW Manual Analyzer* module for analyzing the hardware manual and extracting target information. This component is implemented using Python.

Model Generator is the component for instantiating the meta-model based on the extracted knowledge taken from the *Hardware Knowledge Extractor* and *Software Knowledge Extractor* components. The input of this component is the hardware knowledge in the format shown in 3.3 and the software knowledge in the format shown in 4.3. The output of this component is a model in XMI format. This component includes a module named *Model Instantiator* instantiating this meta-model. This component is implemented in EMF; Java is used for the module

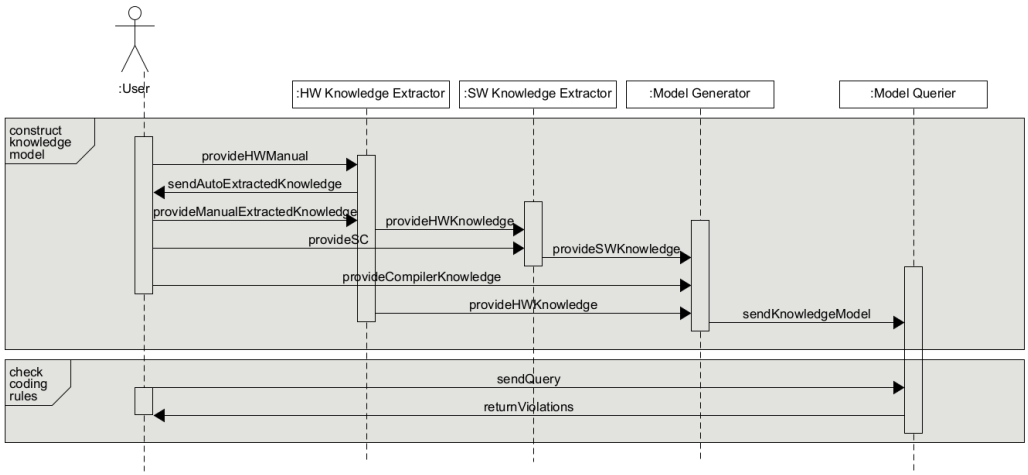

Figure 7.2: Sequence diagram of the whole verification process

Model Generator.

Model Querier component generates and performs queries over the knowledge model. The input of this component is a knowledge model generated from the *Model Generator* components. The output is the verification in the format of Definition 6.1.1. This component includes *Query Executor* module to execute the query. This component is implemented in EMF; the queries are described using QVTo.

7.2 Verification process

Figure 7.2 shows the sequence diagram for verifying systems against the microcontroller-specific coding rules. There are five entities in this process: *User*, *HW Knowledge Extractor*, *SW Knowledge Extractor*, *Model Generator*, and *Model Querier*.

This process has two phases. The first phase is extracting information and instantiating the meta-model using the extracted information. First, the *User* provides a hardware manual to the *HW Knowledge Extractor* to extract hardware knowledge. Then, the *HW Knowledge Extractor* sends back the automatically extracted knowledge to the *User*. The *User* will manually extract the remaining knowledge and send the result to the *HW Knowledge Extractor*. The *SW Knowledge Extractor* receives the full hardware knowledge from *HW Knowledge Extractor* and a source code from the *User*, analyzes the source code, and outputs the software knowledge to the *Model Generator*. The *Model Generator* receives the hardware knowledge from the *HW Knowledge Extractor*, software knowledge from the *SW Knowledge Extractor*, and compiler knowledge from the *User*, instantiate the meta-model and output the knowledge model to the *Model Querier*. The result of the

first phase is a knowledge model of the target system.

The second phase is to query the knowledge model to check for the target coding rules. Specifically, the *Model Querier* receives the knowledge model from the *Model Generator*. Users can send QVTo queries to the *Model Querier* to verify their target coding rules.

Chapter 8

Evaluation

We conducted experiments to evaluate the proposed approach's feasibility and applicability. These experiments were conducted over two kinds of source code: benchmark source code and industrial source code. The benchmark source code is materials prepared by experienced developers of AISW in which this source code contains: several scenarios of code that perform register access (some of them violate coding rules) and common variations to perform register-access, which are frequently used in the industry. This source code was used to evaluate the approach's feasibility and the performance of code patterns when dealing with common variations of register-access in the industry. The industrial source code was a real product of AISW. This source code was used to evaluate the approach's applicability as the source code has a sufficient size and properties of a popular industrial microcontroller-based source code.

Finally, we then discuss the reliability, effectiveness, and applicability of the verification program to other microcontrollers. These discussions can be found in Section 8.3.

8.1 Experiment with benchmark source code

8.1.1 Experiment settings

Table 8.1: Categories of scenario code in benchmark source code

Coding rule category	Check content	Not violate	Violate
read-access	Size of read-access	45	3
non-conditional-case (write-access)	Written value of whole register	26	26
	Written value in several bits	5	5
	Size of write-access	9	9
conditional-case (write-access, one condition)	Written value of whole register	3	1
	Written value in several bits	15	5

We applied the proposed approach over benchmark source code provided by AISW to evaluate the approach in two criteria 1) the feasibility of the proposed approach and 2) the coverage of proposed code patterns in common variations of register-access in the industry.

The benchmark source code consisted of 2,900 lines of code, 48 scenarios of code that related to the read-access, 104 scenarios of code related to the write-access (152 scenarios in total), 39 read-access expressions, 47 write-access expressions, and three violations of read-access coding rules, 49 violations of write-access coding rules. There were 152 scenarios of code in total. The number of register-access expressions was small than the number of scenarios because some scenarios call the same function which contains register-access expressions.

Table 8.1 shows the categories of the 152 scenarios of code. In which, 45 scenarios did not violate read-access coding rules, three scenarios violated read-access coding rules, 40 scenarios did not violate non-conditional-case coding rules, 40 scenarios violated non-conditional-case coding rules, 18 scenarios did not violate conditional-case coding rules, and six scenarios violated conditional-case coding rules. For read-access, the property to be checked is access-size of registers. For write-access, there were three types of properties to be checked: written values of the whole register, written values of several bits, and access-size.

Senior developers of AISW designed the experiments with benchmark source code. In general, these selection criteria in the experiments was to comprehensively evaluate the approach in handling coding rules from all categories, frequently used

register-access expressions and C expressions with various complexity levels. Specifically, the developers selected coding rules so that there is at least one coding rule from each category. In the process of designing the experiments, we also had a concern about different selected sets of coding rules that could result in different experimental results. Hence, we conducted comprehensive experiments with coding rules from all categories. The developers designed violations so that there were violations for all selected coding rules. These violations appeared in various forms of C expressions (i.e., these expressions have different levels of complexity). In other words, these violations were designed to represent various potential appearances of violations in actual industrial source code.

The assumed hardware in this experiment consisted of 110 registers and 152 coding rules (i.e., corresponding with the 152 scenarios of code). Specifically, there were 43 read-access coding rules, 80 non-conditional-case coding rules, and 24 conditional-case coding rules. In this experiment, we assumed that the hardware knowledge was provided in advance. Hence, we did not need to analyze the hardware manual to extract this information.

The assumed compiler in this experiment was a compiler with the name *sampleCompiler*. In this compiler, `unsigned long` had `32` bits with the lower bound being `0` and the upper bound being `18446744073709551615`; `unsigned short` had `16` bits with the lower bound being `0` and the upper bound being `65535`; `unsigned char` had `8` bits with the lower bound being `0` and the upper bound being `255`. The formalized compiler knowledge in this experiment was as follows.

<0, **sampleCompiler**, [<unsigned long, 32, 0, 18446744073709551615>,
<unsigned short, 16, 0, 65535>, <unsigned char, 8, 0, 255>]>

Program analysis tools used in this experiment were Cobra version 3.1[1], Cobra version 3.1, Frama-C version 20.0 (Calcium)[2], CBMC version 5.11[3], and SatAbs version 3.2[4]. The tool for performing natural language processing tasks was NLTK[5]. The tool for knowledge modeling and querying was Eclipse Modeling Tools 2020-09[6].

The machine used in this experiment contained 32 cores CPU and 1.5TB memory for analyzing the hardware manual and source code. For modeling and verifying tasks, the machine for experimenting with representing the knowledge was a computer with 32 GB RAM and an Intel Core i7 CPU.

[1] https://spinroot.com/cobra/downloads.html
[2] http://frama-c.com/download/user-manual-20.0-Calcium.pdf
[3] https://www.cprover.org/cbmc/
[4] https://www.cprover.org/satabs/
[5] https://www.nltk.org/
[6] https://www.eclipse.org/downloads/packages/release/2020-09/r/eclipse-modeling-tools

```
1   for( ; ; ) {
2       switch(main_cnt) {
3       case 0:
4           sample_Exec00();
5           break;
6       [...]
7       default:
8           main_cnt = 0;
9           break;
10      }
11
12      main_cnt++;
13  }
```

Figure 8.1: A missed `for` loop

In this experiment, we first analyzed the source code using the first three steps in Algorithm 3. The fourth step was skipped as no coding rules require the accessed order in this experiment. Next, we modeled the extracted software knowledge, the knowledge of the assumed hardware, and the knowledge of the compiler. Finally, we used the model to verify 152 coding rules using the pre-defined queries.

8.1.2 Experiment results

Table 8.2: Extracting loop objects

	# Total	# Matched expression	# Calculated
For loop	5	4	4
While loop	0	0	0

On extracting loop information: Table 8.2 shows the result of step 1 in Algorithm 3. In this step, four out of five `for` loops in the source code were detected using the code patterns for <*simple-for-pattern*> in Definition 4.4.2. There were five `for` loops and no `while` loops in the source code. The missed loop is shown in Figure 8.1. This loop was out of the scope of the supported loop as the control variable `main_cnt` was updated outside of the loop.

Table 8.3: Extract register-access object

		register-access object status			# explanation
		# confirmed	# maybe	# unknown	
Read-access	Eva plugin	_	34	4	_
	CBMC	38	_	0	38
	SatAbs	x	x	x	x
Write-access	Eva plugin	_	61	12	_
	CBMC	73	_	0	73
	SatAbs	x	x	x	x

On extracting potential register-access expression: Using the code patterns for register access from Definition 4.4.1 to find potential register-access expressions, we detected 60 potential read-access expressions using the code pattern number **(1)** for read-access; 4 potential read-access several bits expressions using the code pattern number **(2-6)**. For write-access, we detected 68 potential write-access expressions using the three code patterns for write-access. The code patterns cover all expected register-access expressions.

On confirming actual register-access object: Table 8.3 shows the result of step 2 in Algorithm 3 in confirming the register-access objects. In this table, "_" mean the tool is not capable of handling this task; "x" means that the tool may be capable but not used. CEGAR/SatAbs was not used in this experiment as the combination of the AI/Eva plugin, and BMC/CBMC is sufficient in this experiment. In Table 8.3, there were four *unknown* read-access objects and 12 *unknown* write-access objects reported by the Eva plugin. However, these *unknown* objects were fully handed by BMC/CBMC later.

Table 8.4: Extracting register-access detail

	Potential accessed bit sets	Potential written value sets	Accessed size
Read-access	4	x	38
Write-access	x	73	73

Table 8.5: Confirming the special written values

	# confirmed	# maybe	# unknown	# explanation
Eva plugin	_	34	39	_
CBMC	73	_	0	73
SatAbs	x	x	x	x

Tables 8.4 and 8.5 show the result of step 3 in Algorithm 3. Table 8.4 shows the result of potentially accessed bits, potentially written values, and access size using the AI/Eva plugin. Specifically, we extracted four potentially accessed bits sets for read-access objects, 73 potential written value sets for write-access objects, and access sizes for 38 read-access and 73 write-access objects.

For 73 registers, there was a special written value that was either valid or invalid written value. Table 8.5 shows the result of checking whether these values were written to these registers in the benchmark source code. AI/Eva plugin and BMC/CBMC were sufficient for confirming the values. Specifically, AI/Eva plugin concluded that 34 special written values *maybe* were written to 34 registers, but *unknown* about the other 39 special values. However, BMC/CBMC confirmed that the 73 special values were actually written to the 73 registers.

Table 8.6: Result of verifying the benchmark source code against register-access coding rules

Coding rule category	Check content	not violated	violated	maybe-Violated	unknown
read-access	Size of read-access	43	5	0	0
non-conditional-case	Written value of whole register	26	26	0	0
	Written value of several bits	5	5	0	0
	Size of write-access	9	9	0	0
conditional-case	Written value of whole register	3	1	0	0
	Written value of several bits	15	5	0	0

```
1  void func_write_16(unsigned long add, unsigned short data){
2      (*((volatile unsigned short*)add)) = (unsigned short)data;
3      return;
4  }
5  void type01_02_SetAll(void){
6
7      unsigned char cnt;
8      const StRegLongData *pt_st;
9      unsigned short tmp_value;
10     unsigned short tmp_mask;
11
12     tmp_mask = 0xFFFF;
13     tmp_value = 0xFFFF;
14     pt_st = type01_02_0E_pt_st;
15     for(cnt = 0 ; cnt < 4 ; cnt++){
16         tmp_mask = (unsigned short)((pt_st->value) >> 16);
17         tmp_value = (unsigned short)((pt_st->value) & 0x0000FFFF);
18         tmp_value = (unsigned short)((~(tmp_value)) & tmp_mask);
19
20         type01_02_func_write_16(pt_st->address, tmp_value);
21         pt_st++;
22     }
23 }
```

Figure 8.2: An unknown case by Eva plugin

On verifying register-access coding rules: Table 8.6 shows the result of verifying the 152 coding rules. In this table, *not violated* means that there was no violation found; *violated*, *maybeViolated*, and *unknown* are defined as in Definition 6.1.1.

For the read-access, we detected five warnings about the read-access size. Three warnings were expected violations. Two remaining warnings were also real violations created unintended by the developers. We could see that it was easy to make unintended bugs when developing microcontroller-based applications for even senior developers. For the write-access, all expected violations were reported as *violated*. It means a warning was triggered for each expected violation; explanations were available for all violations.

8.1.3 Discussion

This experiment showed that the approach is feasible in verifying microcontroller-based systems against the register-access coding rules. Although the benchmark was a small-size source code only, the source code represented different ways for violations to occur. The approach successfully analyzed the source code and detected all expected violations of the target register-access coding rules. We even find violations that senior developers miss.

The coding patterns for register-access expressions could cover register-access methods used in AISW. The proposed code patterns detected all expected register-

access expressions in the benchmark source code, which senior embedded systems developers embedded to represent how they usually do to perform register-access operations. It means the code patterns work well in handling common variations of register-access in the company.

From this experiment, we also found that the combination of multiple program analysis techniques/tools was necessary even for small embedded systems. The reason is that the difficulties of handling these systems not only come from the volume but also complex coding structures used used such as loops, arrays, and pointers. In this experiment, AI/Eva plugin could confirm expressions do not violate the coding rules; however, it could not confirm an expression violate a coding rule. Sometimes this tool failed to provide any information. For example, Figure 8.2 shows an example of these cases where the tool returned in unknown in calculating the register-access in line `2`. In this example, the register-access operations were performed using a loop and pointer to iterate over an array. However, this case could be handed by BMC/CBMC. In this experiment, CEGAR/SatAbs was not required as BMC/CBMC was able to confirm all the actual expected register-access operations. The reason is that the expected violations were apprear within the bound set when executing BMC/CBMC.

8.2 Experiment with industrial source code

To evaluate the applicability of the approach, we applied the approach to handle actual microcontroller-based systems in the automotive industry. There are three sub-experiments in this section. The first sub-experiment was an experiment for analyzing the hardware manual of the target microcntroller. Specifically, we applied the proposed approach for extracting hardware knowledge in Section 3.4 to extract information from the manual. Details of this sub-experiment can be found in Section 8.2.1.

The second experiment was to analyze the industrial source code. Specifically, we applied the proposed approach for extracting software knowledge in Section 4.4 to extract information from the target source code. Details of this sub-experiment can be found in Section 8.2.2.

The third experiment was for generating a data model and checking the target coding rules. Specifically, we applied the meta-model proposed in Chapter 5 and the verification approach proposed in Chapter 6 for this task. Details of this sub-experiment can be found in Section 8.2.3.

8.2.1 Experiment for analyzing hardware manual

Experiment settings

This experiment was conducted by applying the Algorithm 1 to handle the manual of the target microcontroller. The purpose of this experiment was to evaluate the coverage of the patterns in Table 3.2.

The hardware manual of the target microcontroller contained 38 chapters and 2415 pages. There were a total of 641 register groups with 12,149 registers.

We experimented with the approach described in Section 3.4. Specifically, we first convert the manual from PDF format to a text document. Secondly, we extracted physical information from registers. Tasks in extracting registers' physical information included: identifying register subsections and extracting registers' sizes, addresses, values after reset, bit names, and bit accessibility. Thirdly, we extract coding rules for using these registers.

The tools for converting PDF to text were Soda PDF[7] and Microsoft Word[8]. The tool for performing natural language processing tasks was NLTK[9]. The machine used in this experiment contained 32 cores CPU and 1.5TB memory.

Experiment results in extracting physical information of registers

Table 8.7: Result of extracting physical information of registers

Info type	Total	#Correctly extracted	#Wrongly extracted	#Not extracted
Register subsection	641	614	0	27
Access size	641	614	0	27
Access address	614	600	0	14
Value after reset	641	609	1	31
Bit name	641	612	2	27
Bit accessibility	641	614	0	27

Table 8.7 shows the result of extracting physical information of registers. 614 out of 641 register information subsections were detected. In the remaining 27

[7]https://www.sodapdf.com/
[8]https://www.microsoft.com/ja-jp/microsoft-365/word
[9]https://www.nltk.org/

register groups, the accessed address was not described in the hardware manual. Hence, the subsections did not meet the requirement of containing *"\nAddress"*. The registers in these cases were control registers of the microcontroller, which could not be accessed via addresses using standard C programming language. Hence, there was no address for these registers in the manual. However, as our target is source code written in C, the accesses to these control registers are out of our scope.

After identifying register subsections, we extract the information about 614 out of 641 accessed size information was correctly extracted. The remaining 27 accessed size information belonged to the 27 missed register subsections. 600 out of 614 accessed address information was correctly extracted. The remaining 14 accessed address information were cases where the address is referred to somewhere else in the manual. For example, *"Address:\tSee Table 4.3, List of Write-Protection Control Registers."* belonged to these 14 cases.

609 out of 641 value after reset information was correctly extracted. Among 31 missed value after reset information, 27 cases belonged to missed register subsections and 4 cases in which the value after reset refer to somewhere else. One case was wrongly detected because the register group has two possible values after resetting. The sentence to describe the value after reset, in this wrongly detected case, was *"Value after reset:\t0087H (edge detection), 8087H (level detection)"*. *Pattern7* in Table 3.2 detected the first value after reset only.

612 out of 641 bit name information was correctly extracted. Among the remaining 29 cases, 27 missed cases were the cases where accessed size information belongs to the 27 missed register subsections. Two cases were wrongly detected because the bit names were described in multiple lines.

614 out of 641 bit accessibility information was correctly extracted. 27 missed cases were the cases where access size information belongs to the 27 missed register subsections.

Experiment results in extracting coding rules in using registers

Table 8.8: Result of extracting logical information of registers

Category	#Total	#Extracted	#Correctly extracted	#Wrongly extracted
Coding rules related to access-size	641	614	613	1
Coding rules related to accessibility	641	614	614	0
Coding rules related to written values	492	35	35	0

Table 8.8 shows the number of coding rules extracted by applying our approach. For coding rules related to access-size, 613 out of 641 (95.6%) cases were correctly extracted as the coding rules were created based on the information about access size in physical information. For coding rules related to accessibility, 614 out of 641 (95.8%) cases were correctly extracted as the coding rules were created based on the information about accessibility (readable/ writeable/ reserved) in physical information.

For coding rules related to written values, 35 over around 492 coding rules were detected. Program verification techniques could automatically verify 16 out of 35 detected coding rules. For example, *"Set REGISTERNAME33.GROUPBITNAME [5:0] to REGISTERNAME35.GROUPBITNAME [5:0] to 000000B to select physical ch0 ."* belonged to this case. 18 out of 35 detected coding rules; the target bit/ register is not explicitly mentioned. For example, in *"This bit should be set to 0 (REGISTERNAMEnIm signal not output at the beginning of count operation) ."* or *" The program must clear interrupt request flag to 0 ."*, the register names are not stated. There was a detected coding rule where the bit name was not found among extracted registers. The reason was that this bit name belonged to control registers which were out of our scope.

Discussion

Using the patterns in Table 3.2, we extracted the information for 95.8% of registers in one of the most popular microcontroller families used in automotive systems.

95.6% (613/614) of coding rules related to accessed size were correctly extracted. 95.8% (614/641) of coding rules related to the accessibility of the register are correctly extracted. Around 3.3% (16/492) of coding rules related to written values were correctly extracted. Automated program verification techniques could automatically verify the correctly extracted coding rules. By this, a large amount of manual work could be reduced.

The limitation of the patterns in Table 3.2 was that the number of extracted coding rules related to written values was small. However, as we mentioned above, even a small number of coding rules could reduce the number of manual tasks.

8.2.2 Experiment for analyzing C source code

Experiment settings

The target source code in this experiment was a real product of AISW that develops software for car devices. This source code contained 83,006 lines, excluding comments and empty lines.

Similar to the experiment with benchmark source code, program analysis tools used in this experiment were Cobra version 3.1[10], Cobra version 3.1, Frama-C version 20.0 (Calcium)[11], CBMC version 5.11[12], and SatAbs version 3.2[13]. The machine used in this experiment contained 32 cores CPU and 1.5TB memory.

The Eva plugin requires a C pre-processor for use on C files. For pre-processing the source code, a specific commercial compiler (i.e., Green Hills compiler) is necessary. In this experiment, we tried to pre-process the source code using GCC[14]. Several code fragments that GCC could not compile were removed to perform this task. These code fragments were mainly related to compiler-specific features.

In this experiment, we performed the first three steps of the Algorithm 3 to evaluate the following:

- The effectiveness of the combination of selected techniques/ tools
- The supportiveness of knowledge related to loops in extracting knowledge related to register-access.

The result of this experiment was published in [47].

As explained in Algorithm 5, the first step for detecting register-access objects is to detect potential register-access expressions. By applying the code patterns for

[10] https://spinroot.com/cobra/downloads.html
[11] http://frama-c.com/download/user-manual-20.0-Calcium.pdf
[12] https://www.cprover.org/cbmc/
[13] https://www.cprover.org/satabs/
[14] https://gcc.gnu.org/

register-access over the target source code, 1,514 potential read-access expressions and 587 potential write-access expressions were detected. As the target hardware contains 12,149 registers, there were potential 25,525,049 (i.e., 12,149 x 1,514 + 12,149 x 587) register-access objects. The next task was to exclude *confirmedNot* objects, detect *confirmed* and *maybe* objects, and provide explanations for the detected objects. As introduced in Algorithm 5, the checks for potential register-access objects were embedded into assertions. Subsequently, these assertions were executed by three selected tools (i.e., Eva plugin, CBMC, and SatAbs) sequentially. Specifically, we applied the seven steps (i.e., from 2.1 to 2.7) in Algorithm 5 in which the unsolved assertions by the former steps were left to the latter steps.

Firstly, Cobra is used to detect potential register-access expressions using the code patterns for register-access shown in Definition 4.4.2. Subsequently, the detected potential register-access expressions are precisely examined by the Eva plugin of Frama-C, CBMC, and SatAbs as introduced in algorithm **5**. In short, we execute steps with the following settings sequentially: Eva plugin with *NotEqual* assertions attached and loop unwinding not applied, Eva plugin with *NotEqual* assertions attached and loop unwinding applied, CBMC with *NotEqual* assertions attached and loop unwinding applied, Eva plugin with *Equal* assertions attached and loop unwinding not applied, Eva plugin with *Equal* assertions attached and loop unwinding applied, CBMC with *Equal* assertions attached and loop unwinding applied, and finally SatAbs with *NotEqual* assertions attached.

Evaluating many assertions at once was extremely heavy for the Eva plugin, CBMC, and SatAbs. For example, an experiment evaluating 25,525,049 *NotEqual* assertions by the Eva plugin did not complete after three weeks. It is also recommended to execute one assertion per time for CBMC and SatAbs. Hence, we decided to split these assertions into smaller parts so that each part only focuses on one expression. Additionally, the number of assertions executed each time was set differently for each tool. The number of assertions was set to less than or equal to 1000 per run for the Eva plugin but only one per run for CBMC and SatAbs. In this experiment, these executions were performed in parallel as they were independent. Parallel GNU [52] was used to facilitate this parallel process. As the machine used in this experiment contains 64 cores, we set to perform 64 runs in parallel. In Table 8.9, *# assertion/ run* is the number of assertions that are executed per run, and *# run* is the number of executions for each step in this experiment.

Loop unwinding was employed at steps 2.2, 2.3, 2.5, and 2.6 as in Algorithm 5. We applied different schemes for each tool. For applying Eva plugin, unrolling all loops were costly. Additionally, not all loops were relevant to an expression. Hence, only loops closely related to a target expression (i.e., the expression is inside these loops) were unrolled. If this number is available, the number of iterations for a loop is based on the *maxNumOfIteration*. If this number is `unknown`, the *maxNumOfIteration* calculated in the source code is used. For CBMC, there was

Table 8.9: Resource consumed in different steps

Time limit: 48 hours

No.	Step	# assertion/ run	# run	Time (h:m:s)	Memory peak (Kbyte)
2.1	Eva plugin-NotEqual-No loop unwind	<=1,000	27,313	1:08:33	188,488
2.2	Eva plugin-NotEqual-Loop unwind	<=1,000	77	6:27:44	17,788,120
2.3	Eva plugin-Equal-No loop unwind	<=1,000	182	0:59.28	156,380
2.4	Eva plugin-Equal-Loop unwind	<=1,000	182	0:39:13	722,172
2.5	CBMC-NotEqual-d1000	1	2,512	4:49:14	123,521,752
2.6	CBMC-Equal-d1000	1	2,427	3:23:12	122,521,874
2.7	SatAbs-NotEqual-Boom-100	1	221	>48:00:00	3,508,712

a need to specify the bound to unroll every loop. We did not employ the extracted loop objects for unrolling loops with CBMC, as when the number of iterations for loops was large, the executions of CBMC would be out of memory. In this experiment, based on several trials of applying this tool, we selected the setting `-depth 1000` as there were fewer cases of getting out of memory error. This option is depth-based unwinding which uses the number of instructions in the control-flow graph instead of the number of instructions in the source code [53]. For applying SatAbs, a model checker and the number of CEGAR iterations needed to be specified. The selected model checker was Boom [54] as this model checker was selected when SatAbs attended a competition [31]. The number of CEGAR iterations was set to 100 based on our experiment applying this tool to this source code.

As the amount of computation in this experiment might be large, the consumed time might be huge. We set the time limitation for each step as 48 hours in the phase of analyzing the source code.

Experiment results

Table 8.10: Number of detected read-access objects in different steps

(*) 12/15 objects are marked as maybe in step 1. 3/15 objects are newly detected.

No.	Step	confirmed	maybe	unknown	confirmedNot	dead	explanation
2.1	Eva plugin-NotEqual-No loop unwind	-	93	515	4,664,608	13,728,370	-
2.2	Eva plugin-NotEqual-Loop unwind	-	0	515	0	-	-
2.3	Eva plugin-Equal-No loop unwind	0	-	515	-	-	-
2.4	Eva plugin-Equal-Loop unwind	0	-	515	-	-	-
2.5	CBMC-NotEqual-d1000	15(*)	-	512	-	-	15
2.6	CBMC-Equal-d1000	0	-	512	-	-	-
2.7	SatAbs-NotEqual-Boom-100	42	-	-	-	-	42

Table 8.11: Number of detected write-access objects in different steps

(*) 23/70 objects are marked as maybe in steps 1 or 2. 47/70 objects are newly detected.

No.	Step	confirmed	maybe	unknown	confirmedNot	dead	explanation
2.1	Eva plugin-NotEqual-No loop unwind	-	157	26,519	2,633,955	4,470,832	-
2.2	Eva plugin-NotEqual-Loop unwind	-	163	1,741	24,772	-	-
2.3	Eva plugin-Equal-No loop unwind	0	-	1,741	-	-	-
2.4	Eva plugin-Equal-Loop unwind	0	-	1,741	-	-	-
2.5	CBMC-NotEqual-d1000	70(*)	-	1,694	-	-	70
2.6	CBMC-Equal-d1000	0	-	1,694	-	-	-
2.7	SatAbs-NotEqual-Boom-100	102	-	-	-	-	102

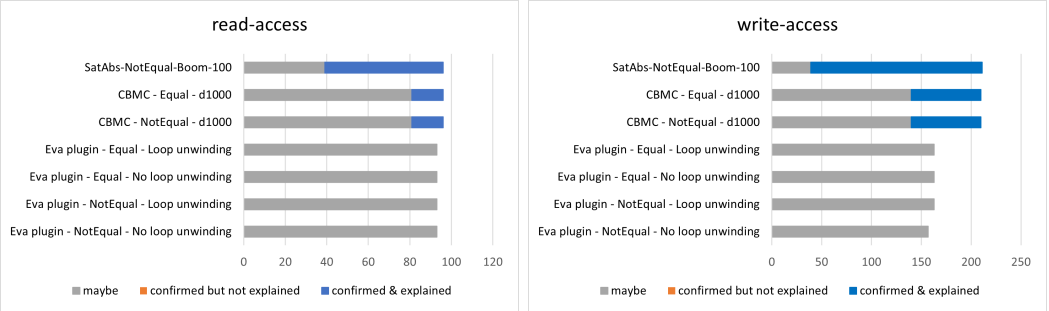

Figure 8.3: Number of detected register-access objects along the extracting process

We detected a total of 208 `for` and `while` loops in the source code, in which 89 loops belong to the supported group. Using the Eva plugin, we calculated the maximum number of iterations for 28 loops. The maximum number of iterations in the source code was 65,536. This means the maximum number of iterations for one loop in the experiment of applying loop unwinding for the Eva plugin is 65,536. It took around 4 seconds to experiment with extracting loop objects.

Figure 8.3 summarizes the final result of detected register-access objects. In this figure, "maybe" means an object was extracted by using the Eva plugin (i.e., $maybe \iff K_{over}, s0 \models \mathbf{EF}\ p$); "confirmed but not explained" means an object was extracted by Eva plugin (i.e., $confirmed \iff K_{over}, s0 \models \mathbf{AF}\ p$); "confirmed & explained" means an object was extracted by CBMC or SatAbs (i.e., $confirmed \iff K_{under}, s0 \models \mathbf{EF}\ p$). We detected 57 `confirmed` objects for read-access and 39 `maybe` objects. The remaining were 512 `unknown` objects. These `unknown` objects belonged to 9 potential read-access expressions. For write-access, we detected 172 confirmed objects and 38 `maybe` objects. The remaining were 1694 `unknown` objects. These `unknown` objects belonged to 35 potential read-access expressions. Explanations were extracted for all `confirmed` register-access objects.

Detailed results of each step can be found in Tables 8.10 and 8.11. In these tables, "-" means that the step could not detect register-access objects with this status. By applying the first step, we could exclude 4,664,608 (25% of potential read-access objects) `confirmedNot` read-access objects, 2,633,955 (36.9% of potential write-access objects) `confirmedNot` write-access objects, 13,728,370 (74.6% of potential read-access objects) `dead` read-access objects, and 4,470,832 (62.7% of potential write-access objects) `dead` write-access objects. This step also detected 93 (0.0005% of potential read-access objects) `maybe` read-access objects and 157 (0.0022% of potential write-access objects) `maybe` write-access objects. The second step was not effective for read-access. However, this step excluded 1741 (0.024% of potential write-access objects) `confirmedNot` write-access objects

and detected 6 (0.00008% of potential write-access objects) `maybe` write-access objects. The third and fourth steps were not effective for both read-access and write-access. The fifth step confirmed and explained that for 15 (0.00008% of potential read-access objects) and 70 (0.00098% of potential write-access objects) write-access objects. The sixth step was not effective for both read-access and write-access. The seventh step confirmed and explained for 42 (0.0002% of potential read-access objects) read-access objects and 102 (0.001% of potential write-access objects) write-access objects.

Table 8.9 shows the time and peak memory consumed for each step. Among these steps, step 5, which applies CBMC with *NotEqual* assertions to detect and explain register-access objects, required the largest amount of memory; step 7, which applies SatAbs with *NotEqual* assertions to explain the detected register-access objects, needed the longest time to perform, and it even not finished within the accepted time (i.e., 48 hours).

Table 8.12: Register-access details

	Potential accessed bit sets	Potential written value sets	Accessed size
Read-access	8	x	608
Write-access	x	1904	1904

Table 8.12 shows the result of potential accessed bits, potential written values, and access size using the Eva plugin. Specifically, we extracted eight potential accessed bits sets for read-access objects, 1904 potential written value sets for write-access objects, and access sizes for 608 read-access and 1904 write-access objects.

Discussion

The experiments show that combining the four techniques effectively detects register-access objects. Specifically, the combination of AI and BMC increased the number of the register-access objects detected. BMC helps to detect 3 more read-access objects and 47 more write-access objects. The combination of BMC and CEGAR effectively confirmed and explained the detected register-access objects. BMC and CEGAR explained different objects, and the number of objects explained by CEGAR was larger than BMC. As a result, most of the detected register-access was explained (59.4% of detected read-access objects, 81.9% of detected write-access objects).

There were remaining 512 `unknown` read-access objects and 1694 `unknown` write-access objects. These objects need to be manually examined. However, these

objects only belong to 9 potential read-access expressions and 35 potential write-access expressions. That means we only needed to manually check 44 expressions in the source code, which was pretty small compared to checking the whole source code.

The knowledge related to loop was supportive in detecting `confirmedNot` register-access objects. As we can see in Table 8.11, by using the knowledge related to loop, 24,772 `confirmedNot` write-access objects were detected at step 2. This result reduced the burden for the latter steps as executing these large number of assertions, especially by CBMC, was heavy. Additionally, 6 `maybe` write-access were detected using the knowledge.

The *NotEqual* assertions were more effective than the *Equal* ones. In fact, the *Equal* assertions did not improve the results in our experiment. The reason is that the *NotEqual* format was more relaxed than the *Equal* format, as in most cases, showing a property was invalid is easier than proving this property is valid. As loops and conditional statements frequently appear in the target source code, the *Equal* format did not perform well in this experiment.

8.2.3 Experiment for generating data model and checking the target coding rules

Experiment settings

We built a knowledge model using the extracted knowledge in Sections 8.2.1 and 8.2.2 using the meta-model described in Section 5.2. Subsequently, we employed the knowledge model to check coding rules in different categories to evaluate the verification framework. Table 8.13 shows the categories of target coding rules.

The tool for knowledge modeling and querying was Eclipse Modeling Tools 2020-09[15]. The machine for experimenting with representing the knowledge is a computer with 32 GB RAM and an Intel Core i7 CPU.

Experiment results

We successfully instantiated the meta-model using the extracted knowledge of the target system. The time for generating the knowledge model in this experiment was around 6 minutes. Table 8.14 shows the result of checking the target coding rules in Table 8.13. In this table, TP means *true positive* which is the number of correctly detected violations; FP means *false positive* which is the number of wrongly detected violations; TN means *true negative*, which is the number of correctly confirmed non-fraud cases; FN means *false negative* which is the number

[15]https://www.eclipse.org/downloads/packages/release/2020-09/r/eclipse-modeling-tools

Table 8.13: Categories of coding rules in the experiment with industrial source code

Category	Coding rule	# Expected violations
Read-access 1O	Do not read bit 4th of register REG1	0
Read-access 1N	Do not read bit 3rd of register REG2	4
Non-cond 1O	Do not write 0x00 to REG3	0
Non-cond 1N	Do not write 0x80010 to REG3	1
Non-cond 2O	Do not set 0xFF to REG4	0
Non-cond 2N	Do not set 0x00 to REG4	1
Cond 1O	If REG5 is 0xCC, REG6 should be set to 0xCC.	0
Cond 1N	Not set REG5 and REG6 to 0xCC at the same time	1
Cond 2O	If REG7 is 0xCC, REG8 should be 0x00.	0
Cond 2N	If REG7 is 0xCC, REG8 should not be 0x2F0002.	1

Table 8.14: Result of verifying industrial source code

Category	# TP	# FP	# TN	# FN	Precision	Recall	Time to query
Read-access 1O	0	0	4	0	-	-	10ms
Read-access 1N	4	0	0	0	-	-	6ms
Non-cond 1O	0	0	1	0	-	-	11ms
Non-cond 1N	1	0	0	0	-	-	11ms
Non-cond 2O	0	**1**	0	0	-	-	12ms
Non-cond 2N	1	0	0	0	-	-	12ms
Cond 1O	0	0	1	0	-	-	8878ms
Cond 1N	1	0	0	0	-	-	8980ms
Cond 2O	0	**1**	0	0	-	-	8840ms
Cond 2N	1	0	0	0	-	-	8852ms
Total	8	2	7	0	0.8	1	-

of missed violation. Regarding the verification result defined in Definition 6.1.1, a TP or an FP means that there is a result with status *violated* or *maybeViolated*. A TN or an FN means that there is a result with status *notViolated* or *unknown*

There were two *false positive* cases in the experiment with the *Non-cond 2O* coding rule and the *Cond 2O* coding rule. The precision value in this experiment was 0.8 as there were two *false positive* cases. The recall was 1 as there were no *false negative* cases. The reason for the wrong warning for checking the *Non-cond 2O* coding rule was that some calculated potential values were not actually written values. Figure 8.4 shows a simplified version of this case. In this figure, `0x020E0004` is the address of REG4. The invalid value `0xFF` was not written to this register. However, there were four possible values for `pt_st->value`, which included `0xFF`. This problem could be solved by considering `0xFF` as a special written value of REG4 and checking whether this value was actually written to this register.

The reason for the wrong warning for checking the *Cond 2O* coding rule was that the register-access object, which wrote `0x00` to REG8, was not detected. Actually, this object was marked with status *unknown*. As this object satisfied the requirement part of the coding rule, the source code did not violate the coding rule. However, a violation was triggered as the verification framework could not detect this object.

```
1  const StRegLongData st_type01_02_0E[4] = {
2      { 0x020E0004, 0xF7317BDE},
3      { 0x020E0008, 0xFF},
4      { 0x020E000C, 0xFFFF5A5A},
5      { 0x020E0010, 0xFFF05A5A}
6  };
7  void setAll(void){
8      unsigned char cnt;
9      const StRegLongData *pt_st;
10
11     pt_st = &st_type01_02_0E[0];
12     for(cnt = 0 ; cnt < 4 ; cnt++){
13         (*((volatile unsigned long*)pt_st->address)) = pt_st->value;
14         pt_st++;
15     }
16 }
```

Figure 8.4: Simplified version of FP for *Non-cond 2O*

Discussion

We successfully applied the meta-model and the verification approach for representing and verifying an industrial system. The meta-model was capable of representing all of the extracted knowledge from the target system.

The precision value in this experiment was 0.8 as there were two *false positive* cases. The recall was 1 as there were no *false negative* cases.

The time query for the model was relatively small: less than one second for read-access and non-conditional-case coding rules and less than 10 seconds for conditional-case coding rules. Hence, the coding rule could be verified interactively after generating the knowledge model. Since verifying coding rules related to register-access usually requires deep analyses of the system, the interactive mode was hard to obtain with existing tools.

As the target source code was a real product in the industry that contains a sufficient size and features of a practical source code, successfully handling this source code is evidence of the applicability of the proposed approach.

8.3 Discussion

8.3.1 Reliability

There are three assumptions in this research. The first assumption is that there is on C source code in the target systems. There are some parts in embedded systems written in other languages, such as assembly. However, the majority are usually implemented in C [19]. Hence, it is still useful to check the parts written in C language. The second assumption is that there are no multiple threads and interruptions. However, we argue that there are cases in that interruptions

do not occur, and our approach can check the target coding rules in this case. Moreover, there is a chance to extend our work to handle multiple threads and interruptions. The third assumption is that all register-access expressions match our code patterns for register-access. It is hard to cover all potential variations of performing register-access using C language. However, in our experiments, the code patterns for register-access worked well with both the benchmark source code and the industrial source code. All expected expressions in the benchmark source code and a sufficient number of expressions in the experiment with the industrial source code (1,514 potential read-access expressions and 587 potential write-access expressions were detected) were detected. Hence, although the three assumptions are applied, our target systems are still significant to handle.

There are two main concerns about the reliability of a verification tool. The first concern is whether this tool misses any bugs. Missing bugs are dangerous, especially for mission-critical systems. In our verification framework, we can ensure that there are no missed bugs in the code expressions that match the code patterns for register-access. The reason is that the expressions will be checked with all the registers in the target microcontroller. Specifically, these expressions are analyzed by multiple sophisticated techniques. This approach performed well in the experiment with the benchmark source code. All expected violations in the benchmark source code were detected. The approach also successfully handled the industrial source code. All expected violations in the industrial source code were detected too. The reason is that the code patterns for register-access covered the popular methods for register-access operations. In the experiment with the benchmark source code, there was no *unknown* register-access objects. In the experiment with the industrial source code, there were 512 *unknown* read-access objects and 1694 *unknown* write-access objects. If the *unknown* objects can be analyzed by the user, there will be no chance for missed bugs in the target expressions.

The second concern is where any reported bugs are false. As the verification framework employs static program analysis tools, there may be cases where false warnings are reported. There are two reasons for false warnings. The first reason is that there is knowledge obtained using over-approximation techniques such as AI, which has not been reconfirmed by under-approximation techniques such as BMC or CEGAR. For example, the first FP case in the experiment with the industrial source code belongs to this case. The second reason is *unknown* register-access objects. The second FP case in the experiment with the industrial source code belongs to this case. False warnings are a common problem with static program analysis tools. There is a trade-off between the rate of FP and FN. As static program analysis tools try to reduce the number of missed bugs, the number of false warnings tends to increase. However, in our framework, the rate of false warnings is expected to be reduced compared with using a single program analysis

tool. The reason is that we first limit our target to a set of special expressions. Subsequently, we apply multiple techniques to precision analyze the values of these expressions. These techniques include over- and under-approximation techniques, where the under-approximation technique is used to re-check the result of the over-approximation technique. This approach helps to reduce the number of false warnings.

There are two bottlenecks of the proposed approach regarding reliability due to technical difficulties. The first bottleneck is the complexity of the hardware manual. This complexity may cause mistakes in the formalization process. Automated solutions can be applied to support the formalization phase. For the target microcontroller in this research, only 16 out of 492 coding rules, which are automatically extracted, can be automatically formalized. However, there is a chance to increase the number of automatically extracted coding rules by introducing better patterns or applying more sophisticated techniques for analyzing manuals.

The second bottleneck is the incompleteness of the proposed code patterns. As the code patterns do not cover all potential register-access expressions, there is a case that register-access expressions are missed. Although the code pattern only covers limited cases, our approach can still be helpful in practice as we focus on the most common patterns. Our experiment with the benchmark source code and the industrial source code shows that these code patterns can cover the commonly used methods for write-access. It also implies that the possibility of undetected violations is low. Additionally, the proposed approach is flexible enough to quickly introduce other different coding styles as new patterns.

8.3.2 Effectiveness

We discuss the effectiveness of the proposed verification approach by comparing this approach with the manual process.

Table 8.15 compares the manual process and the proposed process. In this table, bold parts are automated steps. The two processes contain three phases: *extract coding rules*, *check coding rules*, and *review*. In the manual process, the *extract coding rules* phase is when a developer extracts coding rules from the hardware manual and stores these coding rules as a list in an excel file. In the proposed approach, a part of the extraction process is automated, as explained in Chapter 3.

The *check coding rules* phase contains three steps. The first step is to understand the coding rules. For the manual process, a developer reads the related portions in the hardware manual and profoundly understands the coding rules. For the proposed process, in addition to the task of understanding coding rules, a developer expresses these understandings in the form of formalized coding rules. The second step is to extract source code fragments related to the coding rules. For the manual

Table 8.15: Comparison between the manual process and the proposed process

	Manual process	Proposed approach
Extract coding rules	Extract the coding rules from the hardware manual	**Partially extract the coding rule automatically**
Check coding rules	Deeply understand coding rules	Deeply understand coding rules
		Formalized coding rules
	Extract related fragments of source code	**Extract potentially related fragments of source code**
	Verify related fragments against coding rules	**Analyze the source code**
		Model the hardware and software knowledge
		Issue queries over the modeled knowledge to verify the coding rules
		Execute the queries
Review	Review the result by group discussion	Review unknown register-access objects

Automated parts are in **bold**.

process, a developer looks for these fragments based on his/her knowledge of the coding rules and the target source code. This step is automatically performed for the proposed process using the code patterns for register-access as explained in Section 4.4.2.

The third step is to verify the related fragments against the coding rules. For the manual process, a developer carefully checks each line in the fragments and decides whether this line violates coding rules. This step is automatically performed for the proposed process using multiple program analysis techniques and model-driven engineering techniques. Specifically, required properties in the coding rules are embedded in assertions; subsequently, the assertions are executed using the combination of AI, BMC, and CEGAR. Subsequently, the analysis results of program analysis tools with the hardware and compiler knowledge are modeled using the meta-model explained in Chapter 5. Finally, the modeled knowledge can be queried to check the target coding rules. The task of issuing queries over the model is manual. However, the task of executing these queries is done automatically.

The *review* phase is used for checking the result of the *check coding rules* phase. For the manual process, four to five developers recheck all the results and discuss them to make the final decision. For the proposed process, it is required for a developer to check register-access objects that have `unknown` status.

Compared with the manual process, the proposed process reduces most of the

manual effort. Among the steps in the manual process, the steps which require analyzing source code (i.e., the two last steps in *check coding rules* and the *review* phase) are the heaviest. The first reason is that embedded systems are usually complex and can contain thousands of lines of code. The second reason is that the source code can be frequently updated. In practice, the verification process is conducted before the software release. That means these steps need to be conducted again for every release. On the other hand, steps that require analyzing the hardware manual are usually conducted only once for each microcontroller. We take the source code in the experiment at Section 8.2 as an instance to compare the time consumed between the manual process and the proposed approach. In the manual process, the task of analyzing source code and verifying the source code against the target coding rules, which are three latter tasks in the *check coding rules* phase, the manual process to handle the source code in the experiment at Section 8.2, consumes two man-weeks. In the proposed approach, these steps are mainly automated. The time for analyzing the source code is around 65 hours (i.e., 3925 minutes), the time for building the model is around 6 minutes, and the time for querying is around 83 minutes for 500 coding rules (assume that 10s for one coding rule).

We aim to automate the steps required to analyze source code in the proposed process. As the two last steps in *check coding rules* are automated in the proposed process, the cost of the verification process is dramatically reduced. Although the *review* phase is still required for the proposed process, this phase is less heavy than the manual process. The reason is that instead of checking all the results, the proposed approach only requires checking fragments where corresponding register-access objects get status `unknown`.

Compared with the manual process, the proposed process has an additional manual step (i.e., the formalization step). However, this step can be performed with a little effort as the templates for coding rules are simple and defined beforehand. The formalization step is also valuable to apply for the manual process. This step is helpful if one considers the convenience of discussion as the understanding of coding rules is well documented in a formal format. The formalized coding rules are also reusable for any system built using this microcontroller. Additionally, extracting and modeling hardware knowledge is advantageous in handling multiple source codes. Typically, one microcontroller can be used for multiple systems. Once the hardware knowledge is extracted and modeled, this knowledge can be reused for multiple projects.

Detecting bugs earlier is one of the essential targets in software development. The manual process is unsuitable for this target as it can be performed only a few times (i.e., before a release) because of the high cost. However, the proposed approach is effective in achieving the target as analyzing the hardware manual (i.e., the formalization phase) needs to be done once only at the beginning of conducting

a system; most of the remaining tasks can be done automatically.

8.3.3 Applicability for other microcontrollers

Even though this research focuses on a microcontroller, the proposed approach is applicable to other microcontrollers too. The coding rules of other microcontrollers can fall into two cases. The first case is that the coding rules fall into the defined categories. In this case, the specification language for hardware knowledge can be applied. Performing register-access is a popular task among microcontroller-based systems; our defined specification language for coding rules related to register-access are expected to apply to other microcontrollers. To validate our expectations, we sampled the hardware manuals of three other popular microcontrollers. They are ATmega328P [55], STM32F101xx [56], and PIC16F87XA [57]. In these hardware manuals, we found coding rules which belong to the defined categories below:

1 *"The EEMPE bit must be written to one before a logical one is written to EEPE, otherwise no EEPROM write takes place.",* `Page 21, Section 7` **(ATmega328P) [55]**

2 *"For compatibility with future devices, reserved bits should be written to zero if accessed. Reserved I/O memory addresses should never be written",* `Page 285, Section 29` **(ATmega328P) [55]**

3 *"To enter Stop mode, all EXTI Line pending bits (in Pending register (EXTI_PR)) and RTC Alarm flag must be reset. Otherwise, the Stop mode entry procedure is ignored and program execution continues.",* `Page 60, Chaper 4` **(STM32F101xx) [56]**

4 *"Setting the TPAL and TPE bits at the same time is always safe, however resetting both at the same time can generate a spurious Tamper event. For this reason it is recommended to change the TPAL bit only when the TPE bit is reset.",* `Page 69, Chapter 5` **(STM32F101xx) [56]**

5 *"These registers are reserved; maintain these registers clear.",* `Page 18, Section 2` **(PIC16F87XA)**

6 *"When using the SSP module in SPI Slave mode and SS enabled, the A/D converter must be set to one of the following modes, where PCFG3:PCFG0 = 0100, 0101, 011x, 1101, 1110, 1111.",* `Page 43, Chapter 4` **(PIC16F87XA) [57]**

The second case is that the coding rules of other microcontrollers are out of the defined categories. In this case, one must introduce new templates for describing the new rules. However, the proposed approach is also flexible enough to handle the new template with a reasonable effort. Specifically, the approach is designed as plug-and-play architecture. It is only required to introduce new code patterns and new algorithms for generating assertions that focus on the new categories of coding rules.

The heaviest task in handling other microcontrollers is to handle different styles of writing in hardware manuals. Hardware manuals were usually written in similar

formats, especially for expressing the physical information of hardware. Currently, we apply pattern matching to handle the targeted manuals. Although the patterns in Table 3.2 can not be directly applied to manuals of other microcontrollers, this experiment showed that the heuristic approach seems to work well with this kind of document. As discussed in Section 3.4, there are two kinds of hardware knowledge to be extracted from the hardware manual; physical and logical knowledge of registers. For the physical knowledge, other hardware manuals tend to use table format too, as shown in Figure 3.2. Hence, patterns 1-12 in Table 3.2 may be applied with some modifications. For logical knowledge, manual effort is mainly required. However, as discussed above, the step of analyzing hardware manuals only needs to be done once. Then, the analyzed results can be reused for multiple projects.

Chapter 9

Related works

9.1 Analyzing hardware manual

There are several works on handling natural language documents in software engineering. There are two directions in doing this task.

The first direction is to define grammar with syntactic and semantic categories, manually tag each word in the input text using these categories, and parse the input text based on the grammar [58]. The limitation of this approach is that we need to define the syntactic and semantic categories and manually tag these categories for words. Hence, this approach is hardly acceptable in practice.

The second direction is to apply heuristics based on the characteristic of the target documents [8, 59, 60]. Among these works, the work of S. Chaudhary et al., which also targeted the hardware manual is the closest work to ours. However, this work only handles coding rules related to accessibility. At the same time, our work can handle this type of coding rule and coding rules related to the access size and written values.

9.2 Analyzing and modeling source code information

Extracting information from source code beforehand and using this information for different purposes, such as software verification or understanding, is a well-known approach. Information extraction is one essential task in reverse engineering [61]. As understanding software code is a frequent need in software development, many previous works proposed approaches for analyzing source code and obtaining target information. We discuss the closest works to our work [62, 63, 64]. Akbar et al. [62] provided a comparative evaluation of using information retrieval tech-

niques to explore source code. However, applying information retrieval techniques could not handle the deep analysis of source code that was required in verifying microcontroller-based systems. Hachisu et al. [63] tried to extract information from the source code and store this information in a database. However, the target information was different from our work. While the work of Hachisu et al. tried to get a fine-grained database (i.e., all general information of source code), our work focuses on specific information (i.e., information related to register-access).

On the other hand, the work of Hachisu et al. focused on Java code, while ours focused on C code. Kumar and Krogh [65] proposed an ontology-based knowledge management framework for integrating multiple verification results to eliminate the system-level verification. However, this process was mainly manual. Fehmel et al. [64] automatically extracted and represented the behavior of software drivers in abstract driver finite state machine models. The state machine models are the predicate abstraction of driver features (e.g., API functions or interrupt service routines). While our work focuses on microcontroller-based systems in general, the work of Fehmel et al. focused on driver code. On the other hand, the approach proposed in [64] was only suitable for small source code. When the source code was large, this approach had the following problems: the manual step of identifying variables was heavy, the state explosion problem might occur, and the generated state machine models would be large and hard to be handled by developers who were not familiar with the notations of the models.

There are several solutions for representing the knowledge of microcontroller-based systems. One direct and simple method is to store the knowledge in excel files. Developers of embedded systems use this method. Specifically, the developers manually extract their target knowledge in both source code and hardware manuals; then store this knowledge in excel files; finally, use this knowledge in group meetings to discuss the reliability of the source code. This method is not effective in managing relational knowledge like hardware-dependent knowledge. In addition, the method is not appropriate for automating the verification process. Another method is to store the knowledge in state machine diagrams [66]. In [66], Said et al. extracted the behaviors of embedded software by symbolic execution and represented this behavior in state machines. However, these state machines tended to become complex and hard to understand, while the state machines may store irrelevant information to the target coding rules.

9.3 Verifying microcontroller-based systems

For ensuring the safety of embedded software, while full verification like theorem proving is almost impossible because of its expensiveness, testing is unreliable enough as the coverage is often small compared to many possible cases in a

program. Static program analysis is appropriate since it can automatically check programs against predefined rules. In addition, since this method often employs an over-approximation technique, the possibility of undetected problems is low. Related works also adopted the method to verify the hardware-based properties of microcontrollers.

In [7], Schlich et al. created a tool named [MC]SQUARE for verifying the assembly code of microcontrollers' applications by applying model checking with the support of static analysis. If an error is found, it will be related to the C code using debug information. [MC]SQUARE is a discrete-time, finite-state, mostly explicit CTL model checker for microcontroller assembly code. Users can make propositions about registers, I/O registers, and values of other memory locations, such as C variables. The main problem of this approach was state explosion, as this work could only deal with academic-sized projects. Similarly, [67] and [68] used model checking for verifying embedded software too. However, these works had the same limitation as [7].

By extending a static analysis tool named Goanna, Fehnker et al. have applied static analysis to find bugs in microcontroller software in three categories: incorrect-interrupt-handling check, incorrect-timer-service check, and reserved-bits check [6]. In this research, the CFG of a C program is labeled with the syntactic pattern of interest. Required properties are described manually in the form of CTL formulae. However, this work only focused on a limited set of rules and was hard to be extended to other coding rules.

Similar to [6], Chaudhary et al. introduced a compiler extension named emSPADE [8] which used heuristics to extract rules and apply static analysis to verify reserved-bits, read-only and write-only rules. The advantage of this work was that it was automatic. However, the work only focused on simple rules as they were easier to extract and verify. Another limitation of the work was that it only examined one-line statements and did not consider data-dependent and conditional cases. Embedded programs are often more complex; considering only simple cases is insufficient to ensure these programs' reliability.

In general, previous works tend to create new or extend existing tools, which are heavy tasks. In [7], [67] and [68], modeling various features of a microcontroller is a laborious task. Moreover, the model may need to be adjusted for each microcontroller. Similarly, in [6] and [8], it is necessary to modify the tools directly to handle other rules. This task is very time-consuming. It also requires professional knowledge of both the microcontroller and the verification tool. As the process may lengthen the time-to-market of products, these approaches are not feasible in practice.

Our work combines four existing static program analysis tools to verify a microcontroller's coding rules automatically. This approach is more applicable to the practical situation as it can be easily adapted to check different properties of

Table 9.1: Comparing with existing program analysis tools

Tool name	Pattern-matching-based	Sophisticated analysis	Data query
Flawfinder[12]	yes	no	yes
Polyspace Bug Finder[13]	no	yes	no
CodeQL[14]	no	yes	yes
Our tool	yes	yes	yes

various microcontrollers. Additionally, our target is different from the previous works.

In [7], [6] and [8], they tried to examine the microcontroller's applications against their specifications. Fehnker et al. [6], and Chaudhary et al. [8] aimed at verifying hardware-specific properties. We also target hardware-specific properties, but our coverage is broader than [6] and [8]. Specifically, we can deal with the third rule in [6] and all rules in [8]. We do not handle the two first rules in [6], as these rules do not belong to any category in Table 3.1. However, we cover not only the third rule in [6] and all rules in [8], but other rules. The third rule in [6] and all rules in [8] are only a small portion of our target. Only 18% of the coding rules in the two sections of the hardware manual of the investigated microcontroller are in the same categories as these rules.

Survey about hardware-dependent verification Verification of Hardware Interaction Properties of Software [69] by Ramsay Taylor:

- An analysis process is developed that operates on disassembled executable files and formal specifications of the target platform to produce CSP-OZ formal models of the software's behavior.

9.4 Existing program analysis tools

We proposed a verification tool for handling register-access coding rules. This section discusses our tool with existing program verification tools for C source code. Specifically, we survey popular analysis tools supporting C programs and discuss their applicability to handle specific coding rules like register-access coding rules. As our tool employs pattern-matching-based, sophisticated analysis and data query techniques, we select leading tools in the industry that employ those techniques. For pattern-matching-based techniques, Flawfinder is selected. PolySpace Bug Finder is selected as a sophisticated analysis tool. For data query tools, CodeQL is selected. Table 9.1 compares these tools and our tool regarding techniques used.

Table 9.2: Coding rules supported by existing program analysis tools

Tool name	MISRA C 2012	CERT C	Register-access coding rules	Custom coding rules
Flawfinder	no	no	no	yes
PolySpace Bug Finder	yes	yes	no	no
CodeQL	yes	yes	no	yes
Our tool	no	no	yes	yes

Subsequently, Table 9.2 shows the comparison regarding the handled coding rules. Flawfinder [12] does not directly support MISRA c 2012 and CERT C. However, syntactic-based and partially semantic-based coding rules can be checked by issuing queries over the built database. Hence, using this tool only does not apply to handling our target. PolySpace Bug Finder [13] supports checking coding rule standards such as MISRA C 2012 and CERT C. However, this tool does not support hardware-specific coding rules like register-access coding rules and also does not support custom coding rules. Hence, the tool is not directly applicable to our target. For CodeQL [14], queries can be provided to support MISRA c 2012 and CERT C. This tool is also customizable for new coding rules. However, there is no support customizing the database to represent a source code. Hence, information about used hardware cannot be embedded, and hardware-specific coding rule, like our target, is hardly directly supported.

Chapter 10

Conclusion and future direction

10.1 Conclusion

This research proposed a verification framework to verify microcontroller-based systems against their specific coding rules. There are four sub-goals in proposing this framework: analyzing the hardware manuals to extract hardware knowledge, analyzing the C program to extract software knowledge, modeling the extracted software and hardware knowledge, and verifying systems against the microcontroller-specific coding rules using the extracted software and hardware knowledge.

For the first goal, we proposed a semi-automated approach for analyzing the hardware manuals, extracting and formalizing the information from hardware documents in Chapter 3. We can extract the physical information for 95.8% of registers in one of the popular microcontroller families used in automotive systems. For logical information, 95.6% of coding rules related to accessed size are correctly extracted. 95.8% of coding rules related to the accessibility of the register is correctly extracted. Around 3% of coding rules related to written values are correctly extracted. Automated program verification techniques can automatically verify the correctly extracted coding rules. The registers' information is useful to enable fully automated verification of the coding rules. Although the number of extracted coding rules is small, the approach helps reduce many manual tasks. To precisely describe the hardware knowledge, we propose a specification language for describing the hardware knowledge in Chapter 3. The language can precisely describe the coding rules in the defined categories in Section 3.2.

For the second goal of extracting software knowledge of embedded systems, we proposed an algorithm for combining multiple program analysis techniques (i.e., PM, AI, BMC, CEGAR), which is shown in Chapter 4. The advantage of PM is the flexibility to define patterns to be searched and the high speed even

when scanning large source code. On the other hand, AI, BMC, and CEGAR have the power of sophisticated engines so that they can precisely analyze source code. The experiment results show that the approach effectively extracts the target information. By taking advantage of these techniques, the approach can analyze industrial source code to get software knowledge. Specifically, PM can handle different coding styles and extract syntactic knowledge; AI and BMC can extract semantic knowledge related to register-access, and BMC and CEGAR can explain the extracted knowledge. The extracted knowledge is useful for many tasks in developing hardware-dependent systems, such as understanding the systems and verifying hardware-dependent coding rules. To precisely describe the software knowledge, we propose a specification language for describing the software knowledge in Section 4.3. The specification language for software knowledge can formalize the output of employed program analysis tools.

We proposed a meta-model for the third goal of representing the software and hardware knowledge in Chapter 5. This model can represent knowledge from hardware, compiler, and software and the result of multiple existing extraction tools. Knowledge models of hardware-dependent systems can be generated using the meta-model. Subsequently, these models can be used to verify coding rules related to register-access to the system.

For the fourth goal of handling a large number of variations of coding rules, we proposed an approach for using the knowledge model generated above. This approach is described in Chapter 6. As not all developers are familiar with the query language, we provide predefined queries for the defined categories of coding rules. These queries help to facilitate the usage of the verification approach. The pre-defined queries are explained in Section 6.2.

We implemented a tool based on the proposed verification framework explained in Chapter 7. The tool was used in the experiment with a benchmark source code and an industrial source code in Chapter 8. The result showed that our framework could handle all the scenarios test cases in this benchmark and detected a violation in this industrial source code.

There are two limitations to this research. The first limitation is that we did not handle interruptions and multiple threads. In Section 2.4, we discuss three methods for accessing registers: memory-mapped I/O, which is usually handled by C language, and port-mapped I/O, which usually requires writing in assembly.

The first method is memory-mapped I/O, in which the device registers are mapped to conventional data space. The second method is port-mapped I/O, in which control and data registers are mapped to separate small data spaces. Additionally, several microcontroller families provide special methods of accessing and manipulating the memory via the I/O mapped. The second limitation is that we did not handle assembly code. Although C is the leading choice of embedded systems, there are several situations where implementing the systems in assembly

is necessary, such as speed-critical parts [19]. Typically, embedded systems mix C code with a small portion of assembly code. Assembly code is called from C programs. Assembly is usually used for accessing port-mapped I/O or disabling/ enabling interrupts.

In conclusion, we proposed utilizing advanced techniques in program analysis and model-driven engineering to solve a practical verification problem. Although many innovation techniques proposed in academia have a high potential to apply to practice, the methods of applying these techniques in practice are limited. The reason is that industrial settings are much more complex in comparison with laboratory environments. Despite this difficulty, our work successfully utilized program analysis and model-driven engineering techniques to verify microcontroller-based systems against their specific coding rules. This is a complex problem as the coding rules with numerous variations. Besides the academic contributions, our work has a significant practical contribution as microcontrollers are popularly used in critical systems.

https://scsc.uk/gsn?page=gsn

10.2 Future direction

The first direction is to extend this work to handle assembly code and interruptions. For assembly code, pattern matching may also work well. Several tools may work well for C concurrent programs, such as Deagle [70] or CBMC.

The second direction is that we will deploy the approach in the industrial experiment. Currently, we use a huge computer server for conducting this research. However, there is the case that the environment in practice does not have a large computer as the machine used in our experiments. In this case, the settings of this approach may be changed to adapt to the practical environment. Hence, we will find the optimal setting so the approach can be adapted to multiple environments.

The third direction is to extend the verification framework to handle systems with multiple sub-systems. Currently, we only handle single source code with a single microcontroller. An embedded system, such as an IoT system, may be an integration of multiple devices. Hence, there is room to leverage the verification approach to handle these systems.

Bibliography

[1] Thuy Nguyen, Toshiaki Aoki, Takashi Tomita, and Junpei Endo. Integrating Static Program Analysis Tools for Verifying Cautions of Microcontroller. *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2019-December:86–93, 12 2019. doi: 10.1109/APSEC48747.2019.00021.

[2] Secure use of NFC in medical environments | VDE Conference Publication | IEEE Xplore. URL https://ieeexplore.ieee.org/abstract/document/5755527.

[3] Bill Fleming. Microcontroller units in automobiles. *IEEE Vehicular Technology Magazine*, 6(3):4–8, 9 2011. ISSN 15566072. doi: 10.1109/MVT.2011.941888.

[4] Gerard Le Lann. Analysis of the Ariane 5 flight 501 failure - a system engineering perspective. *Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems*, pages 339–346, 1997. doi: 10.1109/ECBS.1997.581900.

[5] Toyota to recall 1.9 million Prius cars for software defect in hybrid system | Reuters. URL https://www.reuters.com/article/us-toyota-recall-idUSBREA1B1B920140212.

[6] Ansgar Fehnker, Ralf Huuck, Bastian Schlich, and Michael Tapp. Automatic bug detection in microcontroller software by static program analysis. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 5404 LNCS, pages 267–278, 2009. ISBN 3540958908. doi: 10.1007/978-3-540-95891-8{_}26.

[7] Bastian Schlich. Model checking of software for microcontrollers. *Transactions on Embedded Computing Systems*, 9(4), 3 2010. doi: 10.1145/1721695.1721702.

[8] Sandeep Chaudhary, Sebastian Fischmeister, and Lin Tan. Em-SPADE: A compiler extension for checking rules extracted from processor specifications. In *ACM SIGPLAN Notices*, volume 49, pages 105–114. Association for Computing Machinery, 5 2014. ISBN 9781450328777. doi: 10.1145/2597809.2597823.

[9] MISRA. URL https://www.misra.org.uk/publications/.

[10] SEI CERT C Coding Standard - SEI CERT C Coding Standard - Confluence. URL https://wiki.sei.cmu.edu/confluence/display/c/SEI+CERT+C+Coding+Standard.

[11] Rules vs. Recommendations - CERT Secure Coding - Confluence. URL https://wiki.sei.cmu.edu/confluence/display/seccode/Rules+vs.+Recommendations.

[12] Flawfinder Home Page. URL https://dwheeler.com/flawfinder/.

[13] Polyspace Bug Finder - MATLAB & Simulink. URL https://www.mathworks.com/products/polyspace-bug-finder.html.

[14] CodeQL. URL https://codeql.github.com/.

[15] PBE Ong. A Comparison of Static Analysis and Fault Injection Techniques for Developing Robust System Services. *Citeseer*. URL https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=72d9c2aebccf78d436cd07dbcef73d9ddd97070b.

[16] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi, and Yoshiki Higo. An empirical study of security warnings from static application security testing tools. *Journal of Systems and Software*, 158: 110427, 12 2019. ISSN 0164-1212. doi: 10.1016/J.JSS.2019.110427.

[17] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study. *ASIA CCS 2022 - Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security*, pages 602–615, 5 2022. doi: 10.1145/3488932.3497764. URL https://dl.acm.org/doi/10.1145/3488932.3497764.

[18] Dan Saks. Representing and Manipulating Hardware in Standard C and C++.

[19] Daniel W Lewis. Fundamentals of Embedded Software: Where C and Assembly Meet. chapter 5, pages 96–116. Prentice-Hall, 2001.

[20] Ken Thompson. Programming Techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 6 1968. doi: 10.1145/363347.363387. URL https://dl.acm.org/doi/abs/10.1145/363347.363387.

[21] Wojciech Rytter. Algorithms on compressed strings and arrays? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1725:48–65, 1999. ISSN 16113349. doi: 10.1007/3-540-47849-3{_}3/COVER. URL https://link.springer.com/chapter/10.1007/3-540-47849-3_3.

[22] Cobra Static Code Analyzer. URL https://spinroot.com/cobra/.

[23] Gerard J. Holzmann. Cobra: Fast structural code checking (Keynote). In *SPIN 2017 - Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 1–8. Association for Computing Machinery, Inc, 7 2017. ISBN 9781450350778. doi: 10.1145/3092282.3092313.

[24] Patrick Cousot and Radhia Cousot. Abstract interpretation: "A" unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, volume Part F130756, pages 238–252. Association for Computing Machinery, 1 1977. doi: 10.1145/512950.512973.

[25] Patrick Cousot and Radhia Cousot. A gentle introduction to formal verification of computer systems by abstract interpretation. Technical report. URL www.astree.ens.fr/.

[26] Eva, an Evolved Value Analysis. URL https://frama-c.com/fc-plugins/eva.html.

[27] Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring Abstract Interpreters Through State and Value Abstractions. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10145 LNCS:112–130, 2017. doi: 10.1007/978-3-319-52234-0{_}7. URL https://link.springer.com/chapter/10.1007/978-3-319-52234-0_7.

[28] A Biere, A Cimatti, EM Clarke, O Strichman, and Y Zhu. Bounded model checking. 2003. URL https://kilthub.cmu.edu/articles/Bounded_Model_Checking/6603944/files/12094325.pdf.

[29] The CBMC Homepage, . URL https://www.cprover.org/cbmc/.

[30] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1855:154–169, 2000. doi: 10.1007/10722167{_}15. URL https://link.springer.com/chapter/10.1007/10722167_15.

[31] Gérard Basler, Alastair Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl. SatAbs: A bit-precise verifier for C programs (competition contribution). *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7214 LNCS:552–555, 2012. doi: 10.1007/978-3-642-28756-5{_}47.

[32] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. LNCS 3440 - SATABS: SAT-Based Predicate Abstraction for ANSI-C.

[33] About the Unified Modeling Language Specification Version 2.5.1. . URL https://www.omg.org/spec/UML/2.5.1/About-UML/.

[34] Grady Booch, James Rumbaugh, and Ivar Jacobson. The Unified Modeling Language User Guide SECOND EDITION. *Language*, page 496, 2005.

[35] Unified Modeling Language (UML) description, UML diagram examples, tutorials and reference for all types of UML diagrams - use case diagrams, class, package, component, composite structure diagrams, deployments, activities, interactions, profiles, etc. URL https://www.uml-diagrams.org/.

[36] Eclipse Modeling Project | The Eclipse Foundation. URL https://www.eclipse.org/modeling/emf/.

[37] D Steinberg, F Budinsky, E Merks, and M Paternostro. *EMF: eclipse modeling framework*. 2008. URL https://books.google.com/books?hl=en&lr=&id=sA0zOZuDXhgC&oi=fnd&pg=PT23&dq=emf+eclipse+modeling+framework+2nd+edition+pdf&ots=2KLKS_ViFo&sig=zyAAcLo8v-iqJrB0DNHnXUADj1E.

[38] About the MOF Query/View/Transformation Specification Version 1.1. . URL https://www.omg.org/spec/QVT/1.1.

[39] P J Barendrecht. Modeling transformations using QVT Operational Mappings. 2010.

[40] J. W. Backus. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. Preprint. — Software Preservation Group. In *Proceedings of the International Conference on Information Processing*, pages 125–132, 1959. URL https://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf/view.

[41] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *The Computer Journal*, 5(4):349–367, 1 1963. ISSN 0010-4620. doi: 10.1093/COMJNL/5.4.349. URL https://academic.oup.com/comjnl/article/5/4/349/316410.

[42] Amir Pnueli. The temporal logic of programs. *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, 1977-October:46–57, 1977. ISSN 02725428. doi: 10.1109/SFCS.1977.32.

[43] Dov Gabbay. The declarative past and imperative future: Executable temporal logic for interactive systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 398 LNCS:409–448, 1989. ISSN 16113349. doi: 10.1007/3-540-51803-7{_}36/COVER. URL https://link.springer.com/chapter/10.1007/3-540-51803-7_36.

[44] Marco Benedetti and Alessandro Cimatti. Bounded model checking for past LTL. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2619:18–33, 2003. ISSN 16113349. doi: 10.1007/3-540-36577-X{_}3/COVER. URL https://link.springer.com/chapter/10.1007/3-540-36577-X_3.

[45] E. A. Giakoumakis and G. Xylomenos. Evaluation and selection criteria for software requirements specification standards. *Software Engineering Journal*, 11(5):307–319, 1996. ISSN 02686961. doi: 10.1049/SEJ.1996.0041/CITE/REFWORKS.

[46] Thuy Nguyen, Takashi Tomita, Junpei Endo, and Toshiaki Aoki. Integrating pattern matching and abstract interpretation for verifying cautions of microcontrollers. *Software Testing, Verification and Reliability*, page e1788, 8 2021. ISSN 1099-1689. doi: 10.1002/STVR.1788.

[47] Thuy Nguyen, Takashi Tomita, Junpei Endo, Geon-ung Kang, and Toshiaki Aoki. Leveraging hardware-dependent knowledge extraction with multiple

program analysis techniques (in press). In *The 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22)*, Virtual Event, 4 2022.

[48] Edmund M. Clarke, Thomas A. Henzinger, and Helmut Veith. Introduction to model checking. *Handbook of Model Checking*, pages 1–26, 5 2018. doi: 10.1007/978-3-319-10575-8{_}1.

[49] E. Allen Emerson and Joseph Y. Halpern. Sometimes and not never revisited. *Journal of the ACM (JACM)*, 33(1):151–178, 1 1986. ISSN 1557735X. doi: 10.1145/4904.4999. URL https://dl.acm.org/doi/abs/10.1145/4904.4999.

[50] ISO - ISO/IEC 14977:1996 - Information technology — Syntactic metalanguage — Extended BNF. URL https://www.iso.org/standard/26153.html.

[51] The syntax of C in Backus-Naur form, . URL https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm.

[52] Ole Tange and others. Gnu parallel-the command-line power tool. *The USENIX Magazine*, 36(1):42–47, 2011.

[53] The CPROVER Manual, . URL http://www.cprover.org/cprover-manual/cbmc/unwinding/.

[54] Boom. URL https://www.cprover.org/boom/.

[55] Hardware manual of ATmega328P, . URL https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf.

[56] Hardware manual of STM32F101xx, . URL https://www.keil.com/dd/docs/datashts/st/stm32f10xxx.pdf.

[57] Hardware manual of PIC16F87XA, . URL http://ww1.microchip.com/downloads/en/devicedoc/39582c.pdf.

[58] Manish Motwani and Yuriy Brun. Automatically Generating Precise Oracles from Structured Natural Language Specifications. *Proceedings - International Conference on Software Engineering*, 2019-May:188–199, 5 2019. ISSN 02705257. doi: 10.1109/ICSE.2019.00035. URL https://www.ecma-international.org/ecma-262/5.1/index.html#sec-9.6.

[59] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing APIs Documentation and Code to Detect Directive Defects. *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering, ICSE 2017*, pages 27–37, 7 2017. doi: 10.1109/ICSE.2017.11.

[60] Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, China Yongxiang Liu, Lin Tan, and Xiangyu Zhang. C2S: Translating Natural Language Comments to Formal Program Specifications. 2020. doi: 10.1145/3368089.3409716. URL https://doi.org/10.1145/3368089.3409716.

[61] Richard G. Waters and Elliot Chikofsky. Reverse Engineering: Progress Along Many Dimensions. *Communications of the ACM*, 37(5):22–25, 1 1994. doi: 10.1145/175290.175291.

[62] Shayan A. Akbar and Avinash C. Kak. A Large-Scale Comparative Evaluation of IR-Based Tools for Bug Localization. *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, 11:21–31, 6 2020. doi: 10.1145/3379597.3387474. URL https://doi.org/10.1145/3379597.3387474.

[63] Yoshinari Hachisu, Shinichirou Yamamoto, and Kiyoshi Asuga. A CASE Tool Platform for an Object Oriented Language. *IEICE Trans*, 1999.

[64] Thomas Fehmel, Viet Tan Nguyen, Dominik Stoffel, and Wolfgang Kunz. Automatic State Space Analysis for Modeling Untrusted Embedded Device Drivers. *Proceedings - Euromicro Conference on Digital System Design, DSD 2020*, pages 109–116, 8 2020. doi: 10.1109/DSD51259.2020.00028.

[65] Rajesh Kumar and Bruce H. Krogh. Heterogeneous verification of embedded control systems. *Proceedings of the American Control Conference*, 2006: 4597–4602, 2006. ISSN 07431619. doi: 10.1109/ACC.2006.1657445.

[66] Wasim Said, Jochen Quante, and Rainer Koschke. On state machine mining from embedded control software. *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pages 138–148, 11 2018. doi: 10.1109/ICSME.2018.00024.

[67] Eric Mercer and Michael Jones. Model checking machine code with the GNU debugger. In *Lecture Notes in Computer Science*, volume 3639, pages 251–265. Springer Verlag, 2005. doi: 10.1007/11537328{_}20. URL https://link.springer.com/chapter/10.1007/11537328_20.

[68] Michael Weber. An embeddable virtual machine for state space generation. *International Journal on Software Tools for Technology Transfer 2010 12:2*, 12(2):97–111, 3 2010. ISSN 1433-2787. doi: 10.1007/S10009-010-0141-2. URL https://link.springer.com/article/10.1007/s10009-010-0141-2.

[69] Ramsay Taylor. Verification of hardware interaction properties of software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7316 LNCS: 308–322, 2012. ISSN 03029743. doi: 10.1007/978-3-642-30885-7{_}22/COVER. URL https://link.springer.com/chapter/10.1007/978-3-642-30885-7_22.

[70] Fei He, Zhihang Sun, Hongyu Fan, D Fisman, and G Rosu. Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution). *LNCS*, 13244:424–428, 2022. doi: 10.1007/978-3-030-99527-0{_}25. URL https://doi.org/10.1007/978-3-030-99527-0_25.

Publications

Publication in Scholarly Journals

1 Thuy Nguyen, Takashi Tomita, Junpei Endo, and Toshiaki Aoki, "Integrating pattern matching and abstract interpretation for verifying coding rules of microcontrollers." Software Testing, Verification and Reliability 31, no. 8 (2021): e1788, 27 pages, 2021 Dec.

Conference Presentation

International Conference

1 Thuy Nguyen, Takashi Tomita, Junpei Endo, Geon-ung Kang, and Toshiaki Aoki, “Leveraging hardware-dependent knowledge extraction with multiple program analysis techniques”, ACM/SIGAPP Symposium On Applied Computing - Software Verification and Testing track, 10 pages, April 2022.

2 Thuy Nguyen, Takashi Tomita, Junpei Endo, Geon-ung Kang, and Toshiaki Aoki, “Knowledge model for hardware-dependent knowledge” (in preparation).