em-SPADE: A Compiler Extension for Checking Rules Extracted from Processor Specifications

Sandeep Chaudhary
School of Computer Science
University of Waterloo, Canada
skchaudh@uwaterloo.ca

Sebastian Fischmeister
Electrical and Computer Engineering
University of Waterloo, Canada
sfischme@uwaterloo.ca

Lin Tan
Electrical and Computer Engineering
University of Waterloo, Canada
lintan@uwaterloo.ca

Abstract

Traditional compilers ignore processor specifications, thousands of pages of which are available for modern processors. To bridge this gap, em-SPADE analyzes processor specifications and creates processor-specific rules to reduce low-level programming errors. This work shows the potential of automatically analyzing processor- and other hardware specifications to detect low-level programming errors at compile time.

em-SPADE is a compiler extension to automatically detect software bugs in low-level programs. From processor specifications, a preprocessor extracts target-specific rules such as register use and read-only or reserved registers. A special LLVM pass then uses these rules to detect incorrect register assignments. Our experiments with em-SPADE have correctly extracted 652 rules from 15 specifications and consequently found 20 bugs in ten software projects. The work is generalizable to other types of specifications and shows the clear prospects of using hardware specifications to enhance compilers.

Categories and Subject Descriptors D.2.5 [*SOFTWARE ENGINEERING*]: Testing and Debugging; B.5.3 [*REGISTER-TRANSFER-LEVEL IMPLEMENTATION*]: Reliability and Testing—Error-checking

General Terms Bug Detection, Specification, Experimentation, Performance

Keywords Embedded systems, LLVM, static analysis, compiler

1. Introduction

Building embedded systems is time-consuming and it is hard to fix bugs in embedded systems after deployment. Therefore, helping developers build such systems is key to the development process. Because of this reason, automated techniques for bug detection are of great use for embedded software.

Embedded system developers have to write software at low level. This means that they directly program different types of hardware, such as registers, memory, timers, interrupt controllers, I/O controllers, and other peripheral controllers. Developers have to initialize hardware and have to work at the register level. Knowing what to do depends on the processor specification and varies between processors. This variability between processors becomes a likely source of bugs. Bugs might occur for a variety of reasons such as—developers' unawareness of such constraints, insufficient knowledge, human errors, etc. Compiler will be unable to catch these bugs, because they are not syntactical issues.

Embedded system devices possess specifications that state constraints and requirements for these devices. Developers should strictly adhere to the rules and constraints mentioned in the specification to use the device and the associated hardware. As mentioned earlier, because of variability between devices, incorrectly modifying registers is a likely scenario.

Thus, it is imperative to utilize specifications to detect inconsistencies in embedded software. Our analysis demonstrates that it is possible to extract such invariant rules from specifications and use the rules to automatically check for bugs in embedded software.

Thousands of microcontrollers are currently available and each microcontroller has its own specification. Specifications are large and can extend over one thousand pages. Reading the specification before programming a microcontroller is a laborious and tiresome activity. Large volume of information in specifications can cause unintentional mistakes. While analyzing 15 ATMEL AVR specifications, we found 72 registers on average in each specification. With these many registers, one can expect register-related constraints to be many-folds the number of registers.

We manually studied processor specifications to understand what rules are available in processor specifications and how to extract these rules. The focus of this study was on the processor specifications for the AVR family of microcontrollers, which are embedded devices manufactured by ATMEL corporation. To understand the generality of extracting rules from processor specifications, we also studied a specification for NXP semiconductors qualitatively. This qualitatively study demonstrated that rules across different line of devices remain generic to a good extent; thus, it is possible to build a general rule extractor.

The main focus of this project was on extracting and verifying two common types of constraints: (1) access (read/write only), and (2) reserved bits. These rules are in the context of registers. Register bits can be read-only, write-only, or read/write bits. Some bits in registers can be reserved which requires that users do not set reserved bits to one. The implemented rule extractor automatically extracts such rules from processor specifications.

Our tool, em-SPADE, is a static analysis based checker which compares the source code against the extracted rules to automatically detect bugs. Currently em-SPADE checks the validity of register assignments in underlying source code.

The novelty of this paper is that em-SPADE is able to automatically make use of information present in specifications to perform

LCTES '14, June 12–13, 2014, Edinburgh, UK.

http://dx.doi.org/10.1145/2597809.2597823

static analysis. This paper demonstrates that the large volume of the information in specifications can be leveraged for bug detection.

The rule extractor of em-SPADE currently uses simple heuristics to automatically deduce rules from specifications. In the future, we plan to incorporate natural language processing techniques to extract more complex rules from, for instance, English sentences. This will increase the utility of em-SPADE.

We performed experiments for the family of AVR microcontrollers from ATMEL. To test em-SPADE, we collected code from two sources: (1) application notes from ATMEL for their AVR microcontrollers, and (2) github repository. em-SPADE found inconsistencies for several AVR microcontrollers.

We evaluated em-SPADE on 15 specifications and ten open source embedded software projects. em-SPADE extracted a total of 409 read-write only rules with an accuracy of 99.20% and a total of 243 reserved bit rules with an accuracy of 94.88%. We used em-SPADE to check these rules against the source code of the projects, and found 16 warnings related to read-only rules and 4 errors related to reserved bit rules. These bugs are important because writing to a read-only bit or reserved bits might lead to unintended side effects. While such actions might be fine with one revision of the chip, future revisions might alter the use of, for instance, the reserved bits and consequently introduce subtle bugs in previously functionally correct code.

Organization of the rest of the paper is as follows. Section 2 describes the problem statement in detail. Section 3 discusses related work. Section 4 describes the idea and approach of em-SPADE in detail. Section 5 discusses experimental methods. Section 6 discusses the bugs detected by the LLVM [9] checker and the rules extracted by the rule extractor. In Section 7, we describe the performance of em-SPADE. Section 8 discusses some important points about em-SPADE. Finally, in Section 9, we make concluding remarks and discuss future work.

2. Problem Statement

em-SPADE bridges the gap between device specifications and the source code that executes on the device. As described in Section 1, specifications detail requirements and constraints that developers need to follow. However, since the volume of this information is vast, it is probable that developers unknowingly violate some constraints. For example, the specification for ATUC128L3U [27] has 964 pages. If developers want to write some code for the ATUC128L3U, they will need to read the entire document. This tedious task creates a likely scenario of making mistakes.

The goal of em-SPADE is to provide reliability by doing cross-validation of embedded software with the corresponding specification. To demonstrate the usefulness of em-SPADE at a basic prototype level, we are only looking at two types of constraints specified in documents. These constraints are related to value assignment to a different register bits. The first type is about register bits designated as read-only or write-only. If a register bit is read-only and developers write to it, then em-SPADE will issue a warning for this because writing to a read-only register bit might change register behavior. We will use *RO-Writes* to refer to writes to read-only register bits.

The other type is related to reserved bits in registers. If some bit is designated as a reserved bit, developers should not write one to it. If it is present in the source code, then em-SPADE will report an error for this violation. Reserved bits might get some functionality in future versions of the device, so incorrectly writing to them might disrupt some intended future functionality. We will use *Reserved-Writes* to refer to writes to reserved register bits.

In specifications, it is standard to use ':' between two register bits to represent a range. For example, "Bits 4:0" represents five bits i.e., 4, 3, 2, 1 and 0. We will use this notation throughout the paper.

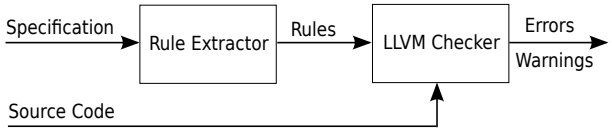

Figure 1. General framework of em-SPADE

In the following, we present one example of each type of rule for the ATmega640/V microcontroller:

1. *"TCCR5C—Timer/Counter 5 Control Register C*
   Bit 4:0—Reserved Bits
   These bits are reserved for future use. For ensuring compatibility with future devices, these bits must be written to zero when TCCRnC is written." [24]
2. *Bits 7, 6, 4 are read-only bits in the Timer/Counter 1 Interrupt Mask Register (TIMSK1).* [24]

Therefore, em-SPADE is addressing the problem of improper value assignment to register bits. The goal of em-SPADE is to provide reliability with respect to two types of probable issues: (1) RO-Writes, and (2) Reserved-Writes. RO-Writes and Reserved-Writes are likely scenarios in the context of microcontroller code. em-SPADE achieves this goal by cross-checking microcontroller code with corresponding rules in the specification. Since specifications are proses in English language, em-SPADE needs to employ heuristics to automatically extract rules of interest. We discuss the heuristics in detail in Section 4. Our aim is to automate the entire toolchain including the rule extractor. Though, several natural language processing techniques are available for extracting rules, em-SPADE is using simple heuristics for now.

3. Related Work

To the best of our knowledge, em-SPADE is the first to automatically analyze processor specifications for automatic bug detection.

Our work consists of two parts: (1) extracting invariants from device specifications, and (2) applying the invariant rules to find bugs in embedded software. Therefore, we discuss work related to rule extraction from natural language documents, and work related to static analysis based bug detection in embedded software.

Fehnker et al. [7] present an automatic bug detection tool which uses static analysis to find bugs in microcontroller software. They examine three types of issues: (1) incorrect-interrupt-handling check; (2) incorrect-timer-service check; and (3) register-to-reserved-bits check. The last check detects similar bugs as em-SPADE. However, the idea of their paper is different from em-SPADE. They manually create rules in the form of CTL formulae for these three types, and their static analysis tool detects bugs based on the rules. Although the bug detection process is partially automatic, automating the rule extraction process is unaddressed, which is exactly the main contribution of em-SPADE.

Dinesh et al. [5] present techniques to extract formal specifications from legal documents, which, are in natural language. They derive CTL specifications from the document which model checking tools can use for verification of models. The specifications are obtained with the use of intermediate semantic representation of different sentences. On the similar line of work, Pandita et al. [17] discuss about extracting formal method specification from natural language text of API documents.

iComment [29] proposes to detect bugs by analyzing comments in the code. They use Natural Language Processing (NLP), statistical and machine learning techniques to analyze comments in source code. The paper presents the novel idea of automatically analyzing

comments to extract programming rules. iComment uses the extracted rules to automatically detect inconsistencies between comments and source code. Based on the analysis, the tool indicates either bugs or bad comments. On the similar line of work, Padioleau et al. [16] discuss the taxonomies and characteristics of comments in operating system (OS) codes. Empirical data presented in the paper shows that comments in OS code are not merely explanations and it is possible to exploit comments for software bug detection.

PR-Miner [11] extracts general programming rules from large software projects and uses them to automatically detect violations in the code. It uses a data mining technique called frequent itemset mining to extract implicit programming rules. AccMon [36] presents automatic detection of memory related bugs using program counter based invariants. Alattin [30] is an alternative pattern mining technique for detecting neglected conditions. RRFinder [32] automatically mines resource-releasing specifications for API libraries. Xie et al. [33, 34] discuss mining techniques for software engineering and program source code data.

Our implementation of rule checker is static which uses LLVM to issue warnings or errors at compile time. The rule checker can be implemented at run-time as well. Csallner and Xi [4], and Smaragdakis and Csallner [21] discuss about combined static and run-time approaches. Engler et al. [6] discuss concepts that lay the foundation of static analysis. The originality of their paper is that they extract the checking information from the code itself and use them to find inconsistencies in the code. The tool can be used to find bugs in source code without any prior knowledge of the system. Hallem et al. [8] describe a framework for performing system specific static analysis.

4. The Framework

This section describes the two components of em-SPADE: (1) Rule extractor, and (2) LLVM checker. The idea of em-SPADE is to extract invariant rules from specifications and use them to detect inconsistencies in embedded software. Therefore, these are related to the two components of em-SPADE respectively. em-SPADE follows the static analysis approach by issuing warnings or errors at compile time to indicate bugs. The automatic approach of extracting rules from specifications is based on heuristics and does not involve any natural language processing.

Figure 1 presents a schematic view of em-SPADE. The first component of em-SPADE, i.e., the rule extractor, takes a microcontroller specification as input and gives the extracted rules in XML format as output. The second component of em-SPADE, i.e., the LLVM checker, takes the XML rules and software code as input and produces errors and warnings as output. We discuss the two components in detail in following subsections.

4.1 Rule Extractor

To gather preliminary data, we first performed an extensive study of four specifications to understand the types of rules. Three of the specifications are for AVR microcontrollers (i.e., ATtiny4 [25], ATmega640/V [24], and ATUC256L3U [27]) and one is for the ARM-M3 based microcontrollers (i.e., UM10360 [28]). The ATtiny4 and ATmega640/V are 8-bit AVR microcontrollers with different sizes of in-system programmable flash. ATUC256L3U is 32-bit Atmel AVR Microcontroller. LPC17xx family are ARM Cortex-M3 based microcontrollers. Section 4.1.1 presents examples of manually extracted rules for ATmega640/V [24] and ARM-M3 processors.

4.1.1 Rules to Extract

While this paper, for now, focuses only on the two mentioned types of rules, the initial study investigated more types of constraints. Consequently, some rules listed below fall outside the category of access type and reserved bit type of rules. These rules show the great potential of generalizing our approach to other types of rules. Some rules from ATmega640/V [24] are:

1. *"XMCRB—External Memory Control Register B; Bit 6:3—Res: Reserved Bits; These bits are reserved and will always read as zero. When writing to this address location, write these bits to zero for compatibility with future devices."*
2. *"ACSR—Analog Comparator Control and Status Register; When changing the ACD bit (i.e., bit 7), the Analog Comparator Interrupt must be disabled by clearing the ACIE bit in ACSR. Otherwise an interrupt can occur when the bit is changed."*
3. *Bits 6:4 in the Clock Prescale Register (CLKPR) are read-only bits.*
4. *"ADCSRB—ADC Control and Status Register B: Bit 7—Res: Reserved Bit; This bit is reserved for future use. To ensure compatibility with future devices, this bit must be written to zero when ADCSRB is written."*
5. *"Bit 0—EERE: EEPROM Read Enable; When the correct address is set up in the EEAR Register, the EERE bit must be written to a logic one to trigger the EEPROM read."*

Some rules from the NXP UM10360 [28] are as follows:

1. *"Reset Source Identification Register (RSID—0x400F C180); 31:4—Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined."*
2. *Bit PLL0STAT in the PLL0 register is read-only and Bit PLL0FEED in the PLL0 register is write-only.*
3. *"PLL1 Status register (PLL1STAT—0x400F C0A8); 31:7—Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined."*
4. *"PLL1 Feed register (PLL1FEED—0x400F C0AC); 31:8 — Reserved, user software should not write ones to reserved bits. The value read from a reserved bit is not defined."*
5. *"External Interrupt Flag register (EXTINT - address 0x400F C140); 31:4—Reserved, user software should not write ones to reserved bits. The value NA read from a reserved bit is not defined."*

Processor specifications follow similar in structure and content across chip vendors. The first specification is for an ATMEL based microcontroller whereas second specification is for microcontrollers from NXP semiconductors. Although these specifications are from different vendors, yet the extracted rules are substantially analogous. Therefore, if it is possible to build a tool to extract rules from AVR microcontrollers, the tool should be able to extract rules from other line of microcontrollers with small or no modifications.

4.1.2 How to Extract the Rules

After this study showed that processor specifications are similar in structure and content, we started creating the automatic rule extractor primarily for the AVR family of microcontrollers. The rule extractor is based on the following observations:

1. The register description layout allows us to get information about the specific bits. Figure 2 shows one such example for Power Control Register 1 (PRR1). Bits 7 and 6 in this register are designated as read-only bits while the rest of the bits are all read and write bits.
2. Register names are in uppercase letters with few numeric characters. Lowercase letters, if any, appear after the first three characters in the name. Also, register name are acronyms with a

Bit	7	6	5	4	3	2	1	0	
(0x65)	–	–	PRTIM5	PRTIM4	PRTIM3	PRUSART3	PRUSART2	PRUSART1	PRR1
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Figure 2. Layout of PRR1 register in ATmega640/V

given acronym description. Timer/Counter Control Register A (TCCR0A) is an example from the ATmega640/V specification. USARTn Control and Status Register B (UCSRnB) is another example from ATmega128 specification.

3. Reserved bits in registers have descriptions. We are able to look up these with keyword searches. For example, EECR register in the ATmega640/V specification has the following description about some reserved bits: "Bits 7:6 — Res: Reserved Bits. These bits are reserved bits and will always read as zero." We have observed this kind of description in all AVR specifications and the NXP UM10360 specification.

Based on these observations, the rule extractor applies the following heuristics to extract the reserved bit and read- and write-only rules:

Heuristics for Reserved Bit Rules – The extraction of reserved bit type of rules starts with identifying the sentences that describe the reserved bits in registers. For this, the rule extractor goes through the sentences and looks for related words such as 'reserved' and 'unused'. Sentences of this type are converted to concrete checkable rules if reserved bit numbers and the register name can both be found. Since the sentence describing the bits is placed as per the bit numbers, it is easy to find the bit numbers given the sentence. The extractor needs only look at the beginning of the sentence, and find the bit number or range of bits. For an example, "Bits 5..0 Res: Reserved Bits - These bits are reserved bits in the ATmega103(L) and will always read as zero." is a description for reserved bit range 5 to 0 in 'SPSR' register in ATmega103. Note that the extractor uses regular expression such as "\\Bit[\\s+s][\\s0-9:,.]*" for matching the bit numbers. This is a 'Boost' [20] based regular expression which requires an extra backslash before any escape sequence involving a backslash such as '\s'.

Extraction of the register name is simple. Unlike bit numbers, the register name might not be available at the beginning of the sentence. However, the name is present in close vicinity of the bit description. Based on our observation, the rule extractor looks at the preceding 100 sentences for the register name. Register names are alphanumeric with all letters in capital. Also, the name is based on the acronym of words which are also available along with the name. For example, "ADC Control and Status Register - ADCSR" in ATmega103. One can observe that the register name 'ADCSR' is an acronym of 'ADC Control and Status Register'. One can use this observation to confirm the validity of the extracted register name.

Heuristics for Read- and Write-only Rules – Extraction of read- and write-only rules is based on the register description layout such as the layout shown in Figure 2. One can observe that each bit in this register is designated as 'R', 'W' or 'R/W'. 'R' represents a read-only bit, 'W' represents a write-only bit and 'R/W' represents a read-write bit. For read-only and write-only rules, we need the bit numbers which are read-only or write-only along with the register name.

The rule extractor uses regular expression matching technique for identifying the register description. It looks for the bit description pattern in the document, and then uses this pattern to find the read-only or write-only category of each bit. The regular expression used is - "[R/W]*\\s[R/W]*\\s[R/W]*\\s[R/W]*\\s[R/W]*\\s[R/W]*\\s[R/W]*\\s[R/W]*". This is again a 'Boost' [20] based regex with the an extra backslash before the escape sequence '\s'. The eight occurrences of 'R/W' in the above regular expression correspond to the eight register bits which are marked as either 'R' or 'W'. This regex identifies register bits from register layouts such as the one in Figure 2. Note that extraction of the register name is same as it is for reserved bit rules.

```
1  <?xml version = '1.0'?>
   <!DOCTYPE rules SYSTEM 'rules.dtd'>
3  <rules>
    <equals>
5     <l>
        <bit_id = 'XMCRB'
7       location = '6' />
      </l>
9     <r>
         0
11    </r>
    </equals>
13 </rules>
```

Figure 3. An example of reserved bit rule in XML

The tool applies a combination of above mentioned heuristics and uses some stop words, such as MCU and AVR, to extract the rules. Specifications of processors are generally in PDF format. em-SPADE first converts it into text format using a free PDF-to-text tool called `pdftotext` [19]. For comparison, we also tried other available tools such as `pdftxt`, `ebook-convert`, `pdf2ps`, `ps2ascii` and `ps2txt`. `pdftotext` is better than all of these in terms of preserving format and information. Once em-SPADE gets the text form of the specification, it reshapes the text by getting rid of empty lines and merging some lines to get the original register description layout. The tool relies on the Boost regular expression libraries [20] to search for patterns and uses C++ containers available in Standard Template Library (STL) to manipulate the intermediate data during rule extraction. The rule extractor writes the rules into a simple text file, which the next module of the tool converts to rules in XML format. The LLVM checker in em-SPADE leverages these extracted rules directly to detect bugs.

4.1.3 Conversion of Rules to XML Format

em-SPADE uses XML to store extracted rules from specifications. The rules XML file is an input to the LLVM checker along with the software code. Figure 3 shows an example of a reserved bit rule in XML. The rule corresponds to bit 6 in External Memory Control Register B (XMCRB) of ATmega640/V [24] microcontroller. In this example, the register name is present in line 6 in Figure 3. The next line lists the location of the reserved bit in the register. Figure 3 shows only the XML expression for the 6th bit of the XMCRB register. Similar expressions can be written for other bits, i.e., 5:3.

4.2 LLVM Checker

This subsection describes the static checker implemented in LLVM [9]. LLVM stands for Low Level Virtual Machine. It is a compiler infrastructure that allows users to perform a variety of optimizations on the source code with the help of LLVM passes. Building a pass creates a shared object, which users can load using the LLVM optimizer tool *opt* [9]. *Opt* can load LLVM specific bitcode files and perform optimizations written in the pass. Using LLVM passes, we can also do static analysis of the source code. The checker of em-SPADE is an implementation of an LLVM pass.

The pass implemented for em-SPADE performs static analysis based on the XML rules file. Since, the rules of interest only have reserved bits and access type of rules, the pass looks at assembly statements, which are assignments to some registers. The pass parses the XML rules using *libxml2* and verifies the validity of assignments. Assignments become store instructions in LLVM intermediate representation (IR) of the program, so the LLVM checker specifically looks at store instructions in the LLVM IR.

If it finds inconsistencies in assignments based on the rules file, it will produce warnings and/or errors. Since, writing one to a reserved bit can cause problems for a future versions of the device, em-SPADE reports reserved bit violations as errors. However, violations of access type of rules produce warnings. It is important to mention that register names in *avr-libc* are present as macros. Therefore, the tool does not find the register names in the IR of the program. This necessitates another small module in em-SPADE which creates a mapping of register names and corresponding macro values. It does the mapping by parsing the specification header file in the library. Header files for all AVR microcontrollers are available in *avr-libc*, which is a part of the *avr-gcc* toolchain. This small module creates this mapping in an simple text file which the LLVM pass reads while going through the IR of the program. These changes do not affect LLVM binaries as the changes are only limited to mapping register name macros from headers files to text files.

em-SPADE compiles the program under test with debug options to get more information about the instructions. Thus, em-SPADE is able to provide sufficient information about the warnings and errors that it reports. This helps developers in locating bugs in the program. Currently, em-SPADE only examines one-line instructions in LLVM intermediate representation, and does not handles cases where writes to reserved or read-only bits are data-dependent or conditional.

In summary, we have implemented em-SPADE in C++ that uses LLVM as back-end for performing static analysis. em-SPADE uses `pdftotext` to parse specification PDF documents in text form. Additionally, it uses C++ standard template library, Boost regex library and *libxml2* library. em-SPADE uses Clang as a front end to get the LLVM bitcode from the source.

5. Experimental Method

This section discusses the experimental method under the following four subsections:

5.1 Subjects & Design

To test technical feasibility and understand how em-SPADE works in practice, we experimented with several AVR processors from ATMEL. To experiment with the rule extractor of em-SPADE, we randomly selected ten specifications for training. After training the rule extractor, we randomly selected another set of five specifications for evaluating the rule extractor of em-SPADE.

For evaluating em-SPADE's capabilities of finding bugs, we looked at application notes [14] and source from ATMEL. Application notes are general application programs for ATMEL based microcontrollers. The application notes span over different domains such as automotive, home appliances, industrial automation, mobile electronics, PC peripherals. em-SPADE analyzed 81 application notes downloaded from ATMEL website. Source code of these application notes contain 50 to 500 lines of code. Apart from this, em-SPADE also analyzed projects for ATMEL AVR microcontrollers available at github.

The 15 specifications in the training and evaluation set cover a wide variety of microcontrollers. They represent three subfamilies: (1) ATmega, (2) ATtiny, and (3) AT90S. These 15 specifications dictate requirements and constraints for 39 AVR microcontrollers. With 39 microcontrollers spanning over three subfamilies, the training and evaluation set becomes representative of AVR microcontrollers. Therefore, the selected set of 15 specifications representing 39 AVR microcontrollers provides a good variability and scale for testing em-SPADE. The 15 specifications are the following:

1. AT90S2313 – 8-bit AVR Microcontroller with 2K Bytes of In-System Programmable Flash
2. AT90S8515 – 8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash
3. ATmega169, ATmega169V – 8-bit AVR Microcontroller with 16K Bytes In-System Programmable Flash
4. ATtiny25/V, ATtiny45/V, ATtiny85/V – 8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash
5. ATtiny24, ATtiny44, ATtiny84 – 8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash
6. ATmega103, ATmega103L – 8-bit AVR Microcontroller with 128K Bytes In-System Programmable Flash
7. ATmega8, ATmega8L – 8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash
8. ATmega128, ATmega128L – 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash
9. ATtiny13, ATtiny13V – 8-bit AVR Microcontroller with 1K Bytes In-System Programmable Flash
10. ATmega48/V, ATmega88/V, ATmega168/V – 8-bit Atmel Microcontroller with 4/8/16K Bytes In-System Programmable Flash
11. ATmega640/V, ATmega1280/V, ATmega1281/V, ATmega2560/V, ATmega2561/V – 8-bit Atmel Microcontroller with 64K/128K/256K Bytes In-System Programmable Flash
12. ATtiny4, ATtiny5, ATtiny9, ATtiny10 – 8-bit AVR Microcontroller with 512/1024 Bytes In-System Programmable Flash
13. ATmega48PA, ATmega88PA, ATmega168PA, ATmega328P – 8-bit AVR Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash
14. AT90S8535, AT90SL8535 – 8-bit AVR Microcontroller with 8K Bytes In-System Programmable Flash
15. ATtiny261/V, ATtiny461/V, ATtiny861/V – 8-bit AVR Microcontroller with 2/4/8K Bytes In-System Programmable Flash

5.2 Apparatus

We performed the experiments on a Lenovo T420 machine which has an Intel Core i5-2520M processor running at 2.50 GHz. It has 4.0 GB RAM memory and is running Ubuntu 12.10 which has 3.5.0-26-generic version of the Linux kernel.

We manually collected the data about the rule extractor by inspecting the generated rules file for each specification. Then, we compared these rules with the manually extracted rules to calcu-

Table 1. Summary of detected bugs

Project	Total	Reserved-Writes	RO-Writes
Optiboot	1	1	0
Libpolulu	3	3	0
AVR064	2	0	2
AVR130	3	0	3
AVR132	3	0	3
AVR312	2	0	2
AVR314	2	0	2
AVR318	1	0	1
AVR319	1	0	1
AVR441	2	0	2
Aggregate	20	4	16

late different metrics. Source codes of AVR based software were downloaded from ATMEL AVR and github websites.

5.3 Measures

We use the following metrics about the extracted rules from different specifications in the training and evaluation set: (1) Actual Total Rule, (2) True Positives (TP), (3) False Positives, (4) False Negatives, (5) Precision, (6) Recall, and (7) F_1 score. Actual Total Rule gives the number of manually extracted rules from the specifications. True positives (TP) are the correct rules reported by em-SPADE. False positives (FP) are the incorrect rules reported by em-SPADE which are not actual rules. False negatives (FN) are the rules that em-SPADE missed to report.

Precision (P) is the fraction of extracted rules that are correct. Precision is defined as:

$$P = \frac{TP}{TP + FP}$$

Recall (R) is the fraction of correct rules that em-SPADE extracts. It is defined as:

$$R = \frac{TP}{TP + FN}$$

F_1 score is a measure of accuracy which takes both precision and recall into account:

$$F_1 = 2 * \frac{P * R}{P + R}$$

F_1 score reaches its best value at one and worst value at zero.

5.4 Procedure

We performed the testing by categorizing the software code according to the specifications in the training and evaluation set. For example, to test microcontroller software for bugs against the ATtiny13 specification, we first extracted the rules from this specification. em-SPADE took the same rules file as input to perform bug detection test on the set of all ATtiny13 software. Therefore, we needed to vary the specification and the LLVM specific bitcode file of the microcontroller software for testing em-SPADE. We generated the LLVM specific bitcode file using the LLVM front-end tool *Clang*.

6. Results

This section discusses the bugs, i.e., errors and warnings, that em-SPADE reported, and the rules that rule extractor of em-SPADE extracted from the AVR family of microcontroller specifications. In the first subsection, we provide a detail description of the errors and warnings that em-SPADE found for some application note projects and github projects.

6.1 Errors and Warnings Detected

em-SPADE found a total of 20 errors and warnings in ten projects. An overall summary of these bugs is available in Table 1. Table 1 lists four columns: (1) Project, (2) Total, (3) Reserved-Writes, and (4) RO-Writes. The first column lists the name of the project. The second column tells the total bugs (errors and warnings) found in the particular project. Third and fourth columns tell the number of Reserved-Writes and RO-Writes bugs found in the project respectively. An elaborate discussion on these two type of bugs is available as follows:

6.1.1 RO-Writes

em-SPADE found 16 RO-Writes type of bugs in eight projects which are application notes published by ATMEL. Although writing to read-only bits might not cause a program to fail, it is bad programming practice and can cause new bugs in future revisions. Therefore, we consider RO-Writes as bugs. These bugs correspond to three specifications, i.e., ATmega169, ATtiny13 and ATtiny24. In all these cases, specifications dictate that the registers have read-only bits that are initialized with 0 but developers incorrectly write 1 to those register bits. The next two paragraphs provide two examples of RO-Writes bugs which em-SPADE found.

Assignment "TIFR1 = 0xFF;" in Main.c in AVR064 project sets all eight bits in TIFR1 register to one. Project AVR064 is intended for the ATmega169 microcontroller and the corresponding specification dictates that bits 7:6 and 4:3 in Timer/Counter1 Interrupt Flag Register (TIFR1) are read-only bits. Hence, setting bits 7:6 and 4:3 to one in TIFR1 register is a violation of the read-only rule and, therefore, the assignment is a bug.

Another example is the assignment "PORTB = 0xFF;" in WakeupTimer/main.c in AVR132. This assignment sets are eight bits in PORTB register to one. The project is intended for the ATtiny13 microcontroller. The ATtiny13 specification dictates that bits 7:6 in Port B Data Direction Register (PORTB) are read-only bits. Hence, setting bits 7:6 to one in PORTB register is a violation of the read-only rule and, therefore, the assignment is a bug.

6.1.2 Reserved-Writes

em-SPADE found one Reserved-Writes type of bug in Optiboot [15]. Optiboot is an optimized bootloader for Arduino [2], and is a quarter of the size of the default bootloader. It allows larger Arduino programs and makes Aurduino programs upload faster. Therefore, it plays an important part as Aurduino bootloader. Optiboot has two target MCUs, i.e., ATtiny84 [26] and ATmega168/V [23]. In both these specifications, bits 7:6 and 4:3 in Timer/Counter1 interrupt flag register (TIFR1) are reserved. However, bynase.c in Optiboot sets all these bits in TIFR1 register to one. This is a violation of the above mentioned reserved bit rule for ATtiny84 and ATmega168/V microcontrollers.

em-SPADE found three Reserved-Writes type of bugs in a library called libpolulu0-avr [12]. The Pololu AVR Library is a collection of support functions for programming AVR-based Pololu products or for using Pololu products with AVRs. It is designed for use with the free *avr-gcc* compiler. Most of the library can also be used together with the Arduino environment. This project targets the following microcontrollers—ATmega48pa [22], ATmega88pa [22], ATmega168pa [22], ATmega328p [22], ATmega48/V [23], ATmega88/V [23] and ATmega168/V [23].

For all these microcontrollers, bits 7:3 in Pin Change Interrupt Flag Register (PCIFR) are reserved. However, in two files in libpolulu project i.e., PololuWheelEncoders.cpp and OrangutanPulseIn.cpp, these bits are set to one as "PCICR = 0xFF;". This is a violation of the reserved bits rule for all seven microcontrollers. Therefore, the mentioned assignment is a bug for libpolulu-avr project. In these specifications, bits 7:6 and 4:3 in Timer/-

Table 2. Data about extracted rules from training specifications

	Reserved bit rules							Read-write only rules						
Specification	Actual	Tool	FP	FN	P (%)	R (%)	F_1	Actual	Tool	FP	FN	P (%)	R (%)	F_1
AT90S2313	15	12	0	3	100.00	80.00	0.89	18	15	0	3	100.00	83.33	0.91
AT90S8515	12	12	0	0	100.00	100.00	1.00	16	13	0	3	100.00	81.25	0.90
ATmega169 ATmega169V	12	11	1	1	91.67	91.67	0.92	46	39	0	7	100.00	84.78	0.92
ATtiny25/V ATtiny45/V ATtiny85/V	16	15	1	1	93.75	93.75	0.94	29	27	1	2	96.43	93.10	0.95
ATtiny24 ATtiny44 ATtiny84	18	18	0	0	100.00	100.00	1.00	29	25	0	4	100.00	86.21	0.93
ATmega103 ATmega103L	14	12	1	2	92.31	85.71	0.89	21	14	2	7	87.50	66.66	0.76
ATmega8 ATmega8L	10	8	1	2	88.89	80.00	0.84	29	20	0	9	100.00	68.96	0.82
ATmega128 ATmega128L	14	13	3	1	81.25	92.86	0.87	30	23	0	7	100.00	76.67	0.87
ATtiny13 ATtiny13V	17	15	0	2	100.00	88.24	0.94	23	22	0	1	100.00	95.65	0.98
ATmega48/V ATmega88/V ATmega168/V	33	28	1	5	96.55	84.85	0.90	45	44	0	1	100.00	97.78	0.99
Aggregate	**161**	**144**	**8**	**17**	**94.74**	**89.44**	**0.92**	**286**	**242**	**3**	**44**	**98.78**	**84.61**	**0.91**

Table 3. Data about extracted rules from evaluation specifications

	Reserved bit rules							Read-write only rules						
Specification	Actual	Tool	FP	FN	P (%)	R (%)	F_1	Actual	Tool	FP	FN	P (%)	R (%)	F_1
ATmega640/V ATmega1280/V ATmega1281/V ATmega2560/V ATmega2561/V	20	19	1	1	95.00	95.00	0.95	54	53	0	1	100.00	98.15	0.99
ATtiny4 ATtiny5 ATtiny9 ATtiny10	24	24	2	0	92.31	100.00	0.96	34	31	0	3	100.00	91.18	0.95
ATmega48PA ATmega88PA ATmega168PA ATmega328P	33	27	1	6	96.43	81.82	0.88	46	42	0	4	100.00	91.30	0.95
AT90S8535 AT90SL8535	17	16	0	1	100.00	94.12	0.97	23	18	0	5	100.00	78.26	0.88
ATtiny261/V ATtiny461/V ATtiny861/V	16	13	1	3	92.86	81.25	0.87	28	23	0	5	100.00	82.14	0.90
Aggregate	**110**	**99**	**5**	**11**	**95.19**	**90.00**	**0.92**	**185**	**167**	**0**	**18**	**100.00**	**90.27**	**0.95**

Counter1 interrupt flag register (TIFR1) are reserved. However, in OrangutanServos.cpp, the assignment "TIFR1 = 0xFF;" sets all the register bits to one which violates the requirement of reserved bits 7:6 and 4:3 in TIFR1 register. Therefore, this statement is also buggy.

The reported 20 bugs in Reserved-Writes and RO-Writes category span across ten AVR projects and 13 different AVR microcontrollers. It is evident that such bugs are prevalent in microcontrollers codes. Therefore, em-SPADE is useful in detecting register assignment bugs in microcontroller software.

6.2 Specification Analysis Results

Table 2 shows the data about extracted rules for the ten specifications in the training set. Similarly, Table 3 shows the data about extracted rules for the five specifications in the evaluation set. Table 2 and 3 list the same metrics for access and reserved-bits types of rules.

Reserved bit rules and read-write only rules are present separately in both tables. Each row in these tables starts with the specification name followed by data about reserved bit rules and read-write only rules. Within reserved bit rules and read-write only rules, the tables list seven entries. Within *reserved bit rules* and *read-write only rules* multicolumns, column "Actual" reports the number of manually extracted rules. Column "Tool" reports the number of correct rules that em-SPADE extracted. In the next two columns, the tables show the number of false positives (FP) and the number of false negatives (FN). The next two columns report precision (P) and recall (R) in percentage. The last column lists the F_1 score which is a collective measure of precision and recall. The range of F_1 score in the tables is from zero to one.

Table 2 shows that the overall precision is 94.74–98.78% and the overall recall is 84.61–89.44% for reserved bit rules and the read-write only rules in the training set. Table 3 shows that the overall precision is 95.19–100.00% and the overall recall is 90.00–90.27% for reserved bit rules and read-write only rules in the evaluation set. The F_1 score of reserved bit rules is 0.92 for both training and evaluation set. The F_1 score of read-write only rules is 0.91 for the training set and 0.95 for the evaluation set. The F_1 score of higher than 0.9 for training and evaluation set indicates that the rule extractor of em-SPADE is accurate, precise, and effective. In addition, it indicates that the rule extractor works accurately for specifications outside of the training set.

Table 2 and 3 show the number of false positives and false negatives for rules extracted from all the specifications. The main reason attributed to both, the false positives and false negatives, is the failure of the heuristics in some cases. In majority of the observed cases, the heuristics fail due to conversion of PDF specifications to text form by `pdftotext`. While converting, `pdftotext` sometimes produces unordered lines or misaligned text for register description layouts. This type of incorrect conversion negatively affects the rule extractor heuristics, which results in false positives and false negatives. In the future, we can use advanced conversion tools or analyze the manufacturers' source files of specifications to reduce false positives and false negatives.

7. Performance

We discuss the performance of em-SPADE in terms of overhead caused by the LLVM pass, overhead caused by the rule extractor. We also discuss the scalability of em-SPADE in with respect to large concatenated specifications.

7.1 Rule Extractor Overhead

Overhead caused by the rule extractor of em-SPADE is not important because rule extraction is a one time task for each processor specification. Once em-SPADE extracts the rules from a particular specification, the LLVM checker can use the same rules file to detect bugs in any software intended for the processor. However, we recorded the one time overhead that the rule extractor causes while extracting the rule. For all the 15 specifications in training set and evaluation set, the rule extractor produced a mean overhead of 4.52 seconds. It took 4.41 seconds of mean CPU time.

7.2 LLVM Checker Overhead

To get data about overhead produced by the LLVM pass implementation, we ran em-SPADE on four projects using rules from the 15 specifications individually. For all these 60 trials, the LLVM checker produced a mean overhead of 1.87 seconds. The checker produced a CPU overhead of 1.79 seconds.

7.3 Scalability of em-SPADE

We tested the scalability of em-SPADE with respect to large specification. To test the scalability of the rule extractor, we combined 15 specs using pdftk [18] to get one large combined PDF file of 3857 pages. The rule extractor completed the extraction in 24.25 seconds. It took 24.10 seconds of CPU time. The mean overhead caused by LLVM checker of em-SPADE in this case was 2.29 seconds. It produced a CPU overhead of 2.22 seconds.

8. Discussion

In this section, we discuss four important points about em-SPADE which provide details about the limitation, effectiveness and generality of em-SPADE. In below subsections, we discuss the following specific points:

8.1 The Need to Modify the Source Code to Get LLVM Bitcode

As mentioned earlier, we have implemented the checker as a LLVM pass. Building the pass creates a shared object which the LLVM optimizer tool can load. The pass works on the LLVM specific binary bitcode file of the source code under inspection. em-SPADE needs to compile the source code using clang to get the LLVM binary. Clang is the front end for LLVM compiler infrastructure. Clang needs 'emit-llvm' option to generate the LLVM specific bitcode output. Most of the projects em-SPADE analyzed were for *avr-gcc* [3] compiler. *Avr-gcc* is a port of GCC which creates binaries for AVR [14] processors. To compile such source codes using clang, we manually need to make some changes such as adding the required header files in the code, providing the path to the include directory of avr library and commenting out a few lines if required. However, in doing so we make sure that none of the changes made put em-SPADE at an advantage in any way as far as finding errors and warnings are concerned.

8.2 False Positives & False Negatives

em-SPADE did not find any false positives while analyzing the embedded software projects. However, since the rule extractor reports false positives, it is probable that em-SPADE may report false positive bugs if the register corresponding to the false rule is assigned some value in the software.

Since the rule extractor is based on heuristics, em-SPADE does not guarantee that it extracts all the rules from specifications. Low recall, specially for ATmega103/ATmega103L and ATmega8/ATmega8L, in Table [1] suggests that em-SPADE misses some actual rules. Inspecting these specifications and the tool heuristics, we found that the limitation comes from `pdftotext` [19]. `Pdftotext` fails to preserve the register description layout in the text format which negatively affects the heuristics. Following the current line of work such as [1], [10], [13], [31] and [35], em-SPADE seeks a balance between false positives and false negatives.

8.3 Incorrect Data in Specifications

The underlying assumption in the context of em-SPADE is that specifications contain correct rules. If there is incorrect data in the specification, then em-SPADE might report false bugs or miss bugs. If developers reference the specification to develop a project, then em-SPADE will not report any bugs because of the consistency between the project code and specification, even though the data in specification is incorrect. However, if developers write projects with the help of their prior knowledge or experience about the device, then em-SPADE will report the bug caused by the inconsistency between project code and the specification. Since specifications act as standard reference guide for developers, it is reasonable to assume their correctness.

8.4 Generality of em-SPADE to other Specifications

Since the rule extractor in em-SPADE makes no specific assumption about ATMEL AVR specifications, em-SPADE should be generalizable to other type of specifications. The heuristics used to extract rules from specifications are applicable to other families of microcontrollers. Our study of NXP LPC17xx microcontrollers gives credence to this belief. Some example rules from these microcontrollers are present in Section 4. One can observe that these rules are similar to the extracted rules from ATmega640/V [24] which have been listed earlier.

9. Conclusion and Future Work

In this paper, we propose a new approach to extract rules from processor specifications automatically and check source code against these rules to detect bugs in embedded systems automatically. We build the prototype em-SPADE, which automatically extracts 652 rules correctly from 15 specifications with precisions of 95.19–100.00% and recalls of 90.00–90.27%. em-SPADE detects 20 bugs in ten ATMEL and AVR software projects automatically, which demonstrates the effectiveness of the approach.

In the future, we plan to employ data mining and natural language processing techniques to extract more complex rules which will generalize em-SPADE and boost the usefulness of em-SPADE. We would accordingly need to enhance our static checker so that it can following type of complex rules:

1. *"To enter any of the three sleep modes, the SE bit in MCUCR must be set (one) and a SLEEP instruction must be executed."*
2. *"The SE bit must be set (one) to make the MCU enter the sleep mode when the SLEEP instruction is executed."*.
3. *"This bit must be set (one) when the WDE bit is cleared, Otherwise, the Watchdog will not be disabled."*
4. *"When changing the ACD bit, the Analog Comparator interrupt must be disabled by clearing the ACIE bit in ACSR."*
5. *" The Stack Pointer must be set to point above $60."*

The idea is to automatically classify such sentences into different categories. Once such classification of sentences is available, it would be easy to extract required information which can be turned to concrete checkable rules.

In addition, we plan to express the extracted rules in LTL formulae to help us categorize the rules. The rules of interest are correctness properties which LTL formulae can express in well defined categories. Once we have categorized the rules, we could convert the LTL formuale directly to XML format. XML is expressive enough to accommodate the extracted rules from specifications. To further improve the precision and recall of the rule extractor, we can use advanced PDF-to-text conversion tools or analyze the manufacturers' source files of PDF specifications. In addition, we plan to port em-SPADE to gcc framework to avoid the issues of generating LLVM IR e.g., mapping of register names and corresponding macro values for generating IR. Another possible future extension is detecting assignment violations involving function calls.

Acknowledgements

This research was funded through grants provided by the Natural Sciences and Engineering Research Council of Canada.

References

[1] P. Anderson. Detecting bugs in safety-critical code. In *Dr. Dobb's Journal*, 2008.

[2] Arduino. `http://www.arduino.cc/`.

[3] Avr-gcc. `http://gcc.gnu.org/wiki/avr-gcc`.

[4] Christoph Csallner and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. In *ISSTA*, pages 245–254. ACM, 2006.

[5] Nikhil Dinesh, Aravind Joshi, Insup Lee, and Bonnie Webber. Extracting Formal Specifications from Natural Language Regulatory Documents. In *Proceedings of the Fifth International Workshop on Inference in Computational Semantics*, 2006.

[6] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 57–72, New York, NY, USA, 2001. ACM.

[7] Ansgar Fehnker, Ralf Huuck, Bastian Schlich, and Michael Tapp. Automatic Bug Detection in Microcontroller Software by Static Program Analysis. In *Proceedings of the 35th Conference on Current Trends in Theory and Practice of Computer Science*, SOFSEM '09, pages 267–278, Berlin, Heidelberg, 2009. Springer-Verlag.

[8] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 69–82, New York, NY, USA, 2002. ACM.

[9] The LLVM Compiler Infrastructure. `http://llvm.org/`.

[10] Holger M. Kienle, Johan Kraft, and Thomas Nolte. System-Specific Static Code Analyses: A Case Study in the Complex Embedded Systems Domain. *Software Quality Control*, 20(2):337–367, June 2012.

[11] Zhenmin Li and Yuanyuan Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 306–315, New York, NY, USA, 2005. ACM.

[12] Pololu AVR Library. `http://www.pololu.com/docs/0J20`.

[13] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically Inferring Multi-variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 103–116, New York, NY, USA, 2007. ACM.

[14] Atmel AVR Microcontrollers. `http://www.atmel.com/products/microcontrollers/avr/default.aspx`.

[15] Optiboot. `https://code.google.com/p/optiboot/`.

[16] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to Programmers—Taxonomies and Characteristics of Comments in Operating System Code. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 331–341, Washington, DC, USA, 2009. IEEE Computer Society.

[17] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 815–825, Piscataway, NJ, USA, 2012. IEEE Press.

[18] Pdftk. http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/.

[19] Pdftotext. http://linux.die.net/man/1/pdftotext.

[20] Boost Regex. http://www.boost.org/doc/libs/1_53_0/libs/regex/doc/html/index.html.

[21] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Proc. 1st International Conference on Tests And Proofs (TAP)*, pages 1–16. Springer, 2007.

[22] ATMEL ATmega48PA/ATmea88PA/ATmega168PA/ATmega328P specification document.

[23] ATMEL ATmega48V/ATmea88V/ATmega168V specification document.

[24] ATMEL ATmega640/V specification document.

[25] ATMEL ATtiny4 specification document.

[26] ATMEL ATtiny84 specification document.

[27] ATMEL ATUC256L3U/ATUC128L3U specification document.

[28] NXP UM10360 specification document.

[29] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*iComment: Bugs or Bad Comments?*/. *SIGOPS Oper. Syst. Rev.*, 41(6):145–158, October 2007.

[30] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294. IEEE Computer Society, 2009.

[31] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. To What Extent Could We Detect Field Defects? An Empirical Study of False Negatives in Static Bug Finding Tools. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 50–59, New York, NY, USA, 2012. ACM.

[32] Qian Wu, Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Iterative mining of resource-releasing specifications. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 233–242. IEEE, 2011.

[33] Tao Xie, M. Acharya, S. Thummalapenta, and K. Taneja. Improving software reliability and productivity via mining program source code. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–5, April 2008.

[34] Tao Xie, Jian Pei, and A.E. Hassan. Mining software engineering data. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 172–173, May 2007.

[35] Wei Zhang, Junghee Lim, Ramya Olichandran, Joel Scherpelz, Guoliang Jin, Shan Lu, and Thomas Reps. ConSeq: Detecting Concurrency Bugs Through Sequential Errors. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 251–264, New York, NY, USA, 2011. ACM.

[36] Pin Zhou, Wei Liu, Long Fei, Shan Lu, Feng Qin, Yuanyuan Zhou, Samuel Midkiff, and Josep Torrellas. AccMon: Automatically Detecting Memory-Related Bugs via Program Counter-Based Invariants. In *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 269–280, Washington, DC, USA, 2004. IEEE Computer Society.