

# Automatic Bug Detection in Microcontroller Software by Static Program Analysis

Ansgar Fehnker<sup>1</sup>, Ralf Huuck<sup>1</sup>, Bastian Schlich<sup>2</sup>, and Michael Tapp<sup>1</sup>

<sup>1</sup> National ICT Australia Ltd. (NICTA)\*  
Locked Bag 6016, University of New South Wales  
Sydney NSW 1466, Australia

<sup>2</sup> RWTH Aachen University\*\*, Embedded Software Laboratory  
Ahornstr. 55, 52074 Aachen  
Germany.

**Abstract.** Microcontroller software typically consists of a few hundred lines of code only, but it is rather different from standard application code. The software is highly hardware and platform specific, and bugs are often a consequence of neglecting subtle specifications of the microcontroller architecture. Currently, there are hardly any tools for analyzing such software automatically. In this paper, we outline specifics of microcontroller software that explain why those programs are different to standard C/C++ code. We develop a static program analysis for a specific microcontroller, in our case the ATmega16, to spot code deficiencies, and integrate it into our generic static analyzer Goanna. Finally, we illustrate the results by a case study of an automotive application. The case study highlights that – even without formal proof – the proposed static techniques can be valuable in pinpointing software bugs that are otherwise hard to find.

## 1 Introduction

Microcontrollers are systems-on-a-chip consisting of a processor, memory, as well as input and output functions. They are mainly used when low-cost and high-reliability is paramount. Such systems can be found in the automotive, entertainment, aerospace and global positioning industry. Since microcontrollers are almost always used in embedded devices, many of them mission critical, a potential re-call is costly. Hence, not only the hardware, but in particular the software running on these microcontrollers has to be reliable, i.e., bug free.

There are a number of formal verification techniques to find bugs or even ensure the absence of them. However, the typically short development cycles for

---

\* National ICT Australia is funded by the Australian Governments Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

\*\* This work was carried out while being on leave to NICTA.

microcontroller-based products made it prohibitive to apply proof-based methods. Model checking and static analysis, which are fully automatic, are in principle suitable for such development environments. Software model checkers such as [1–3] operate on a low level semantic, which allows them to be precise at the expense of speed. Static analysis tools [4–7], in contrast, have been concentrating on a shallower but more scalable and applicable analysis of large code bases [8].

There are a number of obstacles to the application of existing tools to microcontroller software: it is often written in non-standard C, containing assembly statements, direct memory access and custom platform-dependent language extensions [9]. Crucial microcontroller features such as timers and direct memory accesses make model checking in particular challenging, as they require platform-specific hardware models, e.g., for the memory layout, which can result in excessively large state spaces. Common static program analyzers, on the other hand, work on a higher level of abstraction as their main purpose is not to ensure correctness, but to find bugs. If they are able to parse the C dialect, they can easily deal with code base sizes common for microcontrollers. However, commercial static analysis tools typically check for standard deficiencies missing bugs resulting from subtle deviations of the hardware specification.

In this work, we use *Goanna* [10], an industrial-strength static program analyzer that is based on standard model checking technology and can easily be adjusted to include microcontroller-specific checks. *Goanna* works on syntactic abstractions of C/C++ programs. The checks are specified in temporal logic. We demonstrate the strength of this approach by defining and integrating targeted checks for the analysis of the ATMEL ATmega16 microcontroller in a simple and concise manner.

The resulting analysis is fast and precise. This finding is supported by a case study applying *Goanna* to an automotive application. In more than 400 academic C programs, *Goanna* finds about 150 deficiencies, either severe bugs or serious compatibility issues. The rate of false alarms is zero in this case study, that is, all alarms were true alarms. The analysis time for a few hundred lines of ATMEL code is typically below 1 second.

The paper is structured as follows. The next section briefly discusses the particularities of microcontroller code, and what sets it apart from standard ANSI C. Section 3 introduces the static analysis approach via model checking as it is implemented in the tool *Goanna*. Section 4 gives a detailed description of three different rules that we implemented for the ATmega16. Section 5 describes the results that we obtained for an automotive case study. Finally, Section 6 concludes the paper, and discusses future work.

## 2 Why Software for Microcontrollers is Different

C programs for microcontrollers commonly include – besides standard ANSI C language features – compiler-specific constructs, hardware-dependent features, and embedded assembly language statements. One feature that breaks common analysis frameworks is direct memory access, which is a crucial feature, as certain

operations of the microcontroller are controlled by specific registers that are located at fixed memory addresses. An example are I/O registers that are used to communicate with the environment. Most C code model checkers and static analyzers consider direct memory access to be an error [9] because it can lead to defects in an environment with dynamic linking and loading.

One option is to extend standard C code model checkers to cater for microcontroller specific features. This is, however, not an easy task given that the correctness of a program can depend on the underlying hardware layout. Another option is implemented in the [MC]SQUARE tool [11]. It analyzes ATMEGA C code by analyzing the compiled assembly and relating it back to the C code. While this captures all the necessary platform particularities, it also requires to track a large state space, which limits the analysis to certain code sizes. In this paper, we follow the alternative option to amend the static analysis tool Goanna, which bases its checks on temporal logic specifications.

### 3 Static Analysis by Model Checking

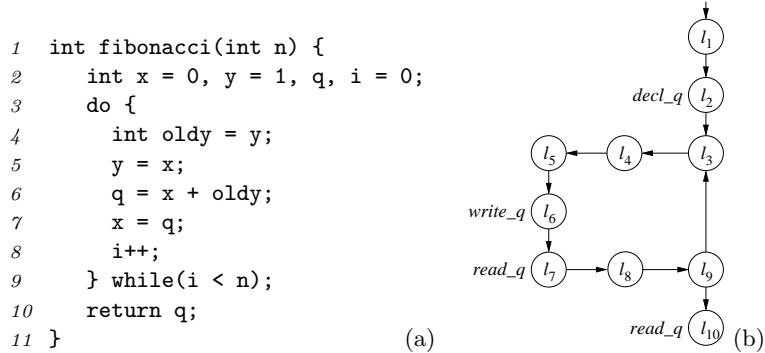
In this work, we use an automata-based static analysis framework that is implemented in our tool *Goanna*. In contrast to typical equation solving approaches to static analysis, the automata based approach [12–14] defines properties in terms of temporal logic expressions over annotated graphs. The validity of a property can then be checked automatically by graph exploring techniques such as model checking. Goanna<sup>3</sup> itself is a closed source project, but the technical details of the approach can be found in [10].

The basic idea of our approach is to map a C/C++ program to its corresponding control flow graph (CFG), and to label the CFG with occurrences of syntactic constructs of interest. The CFG together with the labels can easily be mapped to the input language of a model checker or directly translated into a Kripke structure for model checking. Consider the simple example program `foo` in Fig. 1, which is computing Fibonacci numbers. For example, to check whether variables are initialized before their first use, we syntactically identify program locations that declare, read, or write variables. For variable  $q$  in Fig. 1 (a) we automatically label the nodes with labels  $decl_q$ ,  $read_q$  and  $write_q$ , as shown in Fig. 1 (b). Given this annotated CFG, checking whether  $q$  is used initialized then amounts to checking the following CTL formula.

$$\text{AG } decl_q \Rightarrow (\text{A } \neg read_q \text{ W } write_q) \quad (1)$$

CTL uses the path quantifiers **A** and **E**, and the temporal operators **G**, **F**, **X**, and **U**. The (state) formula **A** $\phi$  means that  $\phi$  has to hold on all paths, while **E** $\phi$  means that  $\phi$  has to hold on some path. The (path) formulae **G** $\phi$ , **F** $\phi$  and **X** $\phi$  mean that  $\phi$  holds globally in all states, in some state, or in the next state of a path, respectively. The *until*  $\phi$ **U** $\psi$  means that until a state occurs along the

<sup>3</sup> <http://nicta.com.au/research/projects/goanna>



**Fig. 1.** (a) Example C program, and (b) annotated control flow graph (CFG). Each node corresponds to one line-of-code for simplicity.

path that satisfies  $\psi$ , property  $\phi$  has to hold. We also use the *weak until*  $\phi \mathbf{W} \psi$ . It differs from the *until* in that either  $\phi$  holds until  $\psi$  holds, or  $\phi$  holds globally along the path. The *weak until* operator does not require that  $\psi$  holds for any state along the paths, as long as  $\phi$  holds everywhere. It can also be expressed in terms of the other operators. In CTL a path quantifier is always paired with a temporal operator. For a formal definition of CTL we refer the reader to [15]. CTL formula (1) means that whenever variable  $q$  has been *declared*, it cannot be *read* until it is *written*, or it is never *read* at all. Note, that the annotated CFG in Fig. 1 (b) satisfies CTL formula (1).

Once patterns relevant for matching atomic propositions have been defined, the CFG will be annotated automatically, and it is straightforward to translate the annotated graph automatically into a Kripke structure, which can then be analyzed by a model checker. Adding new checks only requires to define the property to be checked and the patterns representing atomic propositions. We implemented this framework in our tool Goanna. Goanna is able to handle full C/C++ including compiler-dependent switches for the GNU gcc compiler and uses the open source model checker NuSMV [15] as its generic analysis engine. The run-times are typically in the order of the compilation, i.e., we experience an average overhead of 2 to 3 times the compilation time.

## 4 Codifying the Rules

Microcontroller code is different from common C/C++ source code, and the rules that were developed for large code bases, such as Firefox, have limited applicability in this domain. For example, the standard Goanna tool with its predefined properties does not produce any warnings for the microcontroller-specific case study presented in Section 5.

This section describes how to define platform-specific properties that look for common deficiencies in microcontroller code. Three aspects of microcontroller code are especially prone to error: the correct handling of interrupts, the correct

```

133 //ISR for Timer0 Overflow
134 SIGNAL (SIG_OVERFLOW)
135 {
136     cli();                //deactivate all Interrupts
137     outp(0x00,TCCR0);    //stop Timer0
138
139     mode++;
140     if(mode > 4) mode = 0;
141
142     outp(0x00,TCNT0);    //timer0 reset
143     outp(0x04,TCCR0);    //start Timer0 with prescaler = 256
144     sei();                //activate all Interrupts
145 }

```

**Fig. 2.** Example of a non-interruptible routine violating the interrupt-handling check.

call to and the correct use of timers, and the use of special function registers. For this paper, we have chosen to develop rules specific for the ATMEL ATmega16, to illustrate the approach, but the rules can be extended and changed to fit other platforms as well.

#### 4.1 Incorrect-Interrupt-Handling Check

A common cause of bugs in microcontroller code is the incorrect disabling and enabling of interrupts in interrupt service routines (ISRs). The ATmega16 provides two types of ISRs. The first type disables by default all interrupts at the beginning of the ISR, and enables them by default when it has been handled. The programmer should at no point in the ISR enable or disable any interrupt. The second type of ISRs requires from the programmer to pair enabling and disabling of interrupts. He has to disable them before he can enable them. Unlike other microcontrollers, the ATmega16 does not provide interrupts with priorities, and typically also not that ISRs can be preempted.

To deal with interrupt handling we define syntactic patterns for the following labels:

- *signal* is the label for the entry to an ISR that automatically disables interrupts when entering and enables interrupts when leaving. Interrupts should not be enabled or disabled in this routine.
- *interrupt* is the label for the entry to an ISR that does not disable interrupts when entering and does not enable interrupts when leaving. If someone disables interrupts in this handler, he should enable them afterwards.
- *cli* is the label for register assignments that disable all interrupts.
- *sei* is the label for register assignments that enable all interrupts.
- *fnend* is the label for the end of the routine.

Note, that the preprocessor replaces the commands `cli` and `sei` by register assignments, i.e., our patterns work on these assignments. Given the labels defined above, we define the following rules for the scope of the ISR:

- The rule that ISRs with the attribute *signal* should not enable or disable interrupts at all is expressed in CTL formula (2).

$$\mathbf{AG}(\text{signal} \Rightarrow (\mathbf{AG}\neg(\text{cli} \vee \text{sei}))) \quad (2)$$

- Other ISRs, with the attribute *signal*, have to disable and enable interrupts themselves. If they do, they have to first disable the interrupts, i.e., they cannot enable them, unless they have disabled them earlier. This is expressed in CTL formula (3).

$$\mathbf{AG}(\text{interrupt} \Rightarrow (\mathbf{A}\neg\text{sei}\mathbf{W}\text{cli})) \quad (3)$$

We use the weak until operator  $\mathbf{W}$  to denote that it is acceptable to never disable interrupts.

- If interrupts are disabled, they should always be enabled before the routine leaves the ISR. This is encoded in CTL formula (4)

$$\mathbf{AG}(\text{cli} \Rightarrow (\mathbf{A}\neg\text{fend}\mathbf{W}\text{sei})) \quad (4)$$

- And finally, interrupts should not be enabled twice, without being disabled in-between, and vice versa, not disabled twice, without being enabled in-between. This is encoded in formulae (5) and (6).

$$\mathbf{AG}(\text{cli} \Rightarrow \mathbf{AX}(\mathbf{A}\neg\text{cli}\mathbf{W}\text{sei})) \quad (5)$$

$$\mathbf{AG}(\text{sei} \Rightarrow \mathbf{AX}(\mathbf{A}\neg\text{sei}\mathbf{W}\text{cli})) \quad (6)$$

The temporal operator  $\mathbf{AX}$  is used in the last two CTL formulae because each state labelled *cli* trivially violates  $\mathbf{A}\neg\text{cli}\mathbf{W}\text{sei}$ . This operator states that **after** a state labelled *cli* there should not follow another state labelled *cli*, unless a state labelled *sei* has been encountered earlier along the same path.

*Example.* Figure 2 shows a routine with attribute *signal*, which means that it is not interruptible. Interrupts are disabled before the routine is entered. Use of `sei()` in line 144 opens a window for other routines to interfere, and to corrupt the stack. This ISR does not satisfy (2), and this bug will be flagged as an error.

## 4.2 Incorrect-Timer-Service Check

The ATmega16 has three timers. The programmer can define different ISRs for these timers. It can be syntactically checked which timer a service routine should refer to. Two of the three timers have two configuration registers, and the other one four. When a routine uses one type of timer, the programmer should not change the configuration registers of other timers. For each timer *i*, we introduce the following labels:

- *timer<sub>i</sub>* is the label for the entry to a routine that should use timer *i*.
- *config<sub>i</sub>* is the label for an assignment to registers that modifies the configuration registers of timer *i*

For instance, timer 0 is used correctly if CTL formula (7) holds. We include analogous checks for timers 1 and 2.

$$\mathbf{AG}(\text{timer}_0 \Rightarrow (\mathbf{AG}\neg(\text{config}_1 \vee \text{config}_2))) \quad (7)$$

```

18 SIGNAL (SIG_OVERFLOW2)
19 {
20     TCNT0 = START_VALUE; // reload timer with initial value
21     ++g_ticks;
22 }

```

**Fig. 3.** Example of a routine violating the interrupt-service check.

```

63 void timer_init(void)
64 {
65     TCCR1A = 0x00; // no compare/capture/pwm mode
66     TCCR1B = 1 << CS12 | 1 << CS11; // External clock source on T1 pin.
67     // Clock on falling edge.
68 }

```

**Fig. 4.** Example code violating the reserved bit property.

*Example.* Figure 3 shows an example of an ISR that violates this check. At line 20, timer 0 is assigned an initial value, but the routine was triggered by timer 2.

### 4.3 Register-to-Reserved-Bits Check

The ATmega16 data sheet defines for each register the use of its bits, but it also defines which bits are reserved and should not be used. The Global Interrupt Control Register (GICR), for example, is a register used to store which interrupts have been enabled. For the ATmega16 five bits are used; the three most significant bits to enable or disable external interrupts, and the two least significant bits are used to handle the so called Interrupt Vector table. The three remaining bits are reserved. Bits in registers may be reserved because they are used internally, by future extensions of the ATmega16, or by other microcontrollers of the same family, such as the ATmega128. If a program modifies reserved bits while running on an ATmega16, it might not cause any unexpected behavior. Such deficiencies may remain undetected by any amount of testing on the ATmega16, but only once the program is deployed to a different or extended platform they cause problems.

The data sheet detailing the reserved bits of the ATmega16 can be viewed as a map from addresses to reserved bits. The reserved bits are akin to a mask. For the ATmega16 there are 14 registers that have reserved bits. It is easy to adapt this mapping from registers to reserved bits for other platforms.

Given such a mapping from registers to masks, any assignment to a register that matches a mask is a potential deficiency. Checking if an assignment accesses a certain register, say GICR, is a syntactic check. Checking if the assigned value matches the associated mask of reserved bits is also a syntactic check. If this happens it is flagged as a violation of the register-to-reserved-bit check. The check can be formulated as an invariant  $AG \neg assign\_reserved_i$ , where  $assign\_reserved_i$  is a proposition that can be syntactically defined on the AST for a mask  $i$ .

*Example.* Figure 4 shows an example microcontroller code that violates the reserved-bit property. At line 66 reserved bit 5 of the Timer/Counter1 Con-

trol Register B (TCCR1B) is set to zero. If the programmer had used `TCCR1B = TCCR1B | 1 << CS12 | 1 << CS11`; instead, or set the bits individually, the value of bit 5 would have been preserved. Goanna flags this potential compatibility issue, warning that the programmer should only write bits that are actually used by the ATmega16 microcontroller.

## 5 Application to an Automotive Case Study

This section presents a case study that uses the additional checks for the ATmega16 discussed in the previous section. We have implemented these checks in Goanna. In this case study, we applied Goanna to different academic solutions developed in a lab course at the RWTH Aachen University. In this lab course, students had to solve an automotive real-time problem, namely a four channel speed measurement with a CAN bus interface. The course focussed on the differences between reconfigurable hardware and CPU-based systems, and the students were required to implement one solution using a Complex Programmable Logic Device (CPLD) and one solution using a microcontroller. For latter, an ATMEL ATmega16 microcontroller in combination with a PHILIPS SJA1000 stand-alone CAN controller was used. The aim was to provide a solution that could be integrated into an experimental vehicle. Each group had to develop and implement their own control strategies. The final submission of each group at the end of the lab course had to pass an elementary acceptance test to obtain a certain level of quality in all versions. The details of this process, and more general, of the lab course and its educational aims are described in [16].

The same code base was used in [17] to demonstrate the capabilities of the model checker [MC]SQUARE. This work selected solutions of three groups and subjected them to a thorough analysis. The specification formalized three aspects of a correct implementation: (1) the program should not exhibit stack-collisions, (2) it should ensure that the measured speed is within given bounds, and (3) the unit should eventually produce a defined output if the input exceeds a certain threshold. The first two properties are safety properties, while the latter is a liveness property. Even though the students did not follow any guidelines to make model checking easier, the model checker could be applied with no or little conditioning of the source code. The run times range from a few seconds to find a counterexample, up to 9 hours to show correctness.

### 5.1 Code Base

For this paper, we use the C code that was developed for the ATmega16, and is thus aimed at the *avr-gcc* compiler. There are a total of 475 files, of which 439 are proper C, i.e., these files include all header files and can be parsed. Files that cannot be parsed were mostly scratch files, and in some cases not even C; one file, for example, is just a technical ASCII illustration of the experimental setup. Of the 439 proper C files, Goanna is able to analyze 431, the others cause an exception in the interface between Goanna and the front-end parser. The

cause for this exception needs further investigation. The code is organized into 24 projects. Each project contains the final version of a student group, and for some groups also intermediate versions.

The entire code base of 431 files consists of 97527 LoC (lines-of-code) before, and 203638 LoC after preprocessing. The average size after preprocessing is 472 LoC, while the size of the largest file is 1489 LoC; 98% of all files are smaller than 1000 LoC. Each file consists on average out of 9 functions, while the maximal number of functions is 58; 99% of all files have less than 25 functions. The code base covers a whole range of programs, from well structured code to monolithic "spaghetti" code, which makes it a suitable testbed for software analysis techniques.

## 5.2 Implementation Notes

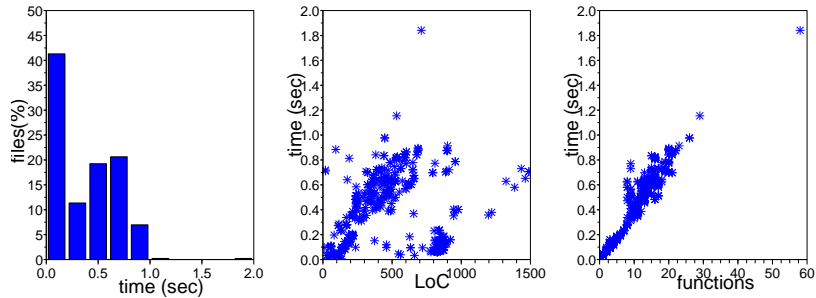
The source code analysis tool Goanna is implemented in Ocaml. It uses a commercial front-end to parse the source code and the symbolic CTL model checker NuSMV as back-end. The front-end supports most common C/C++ compilers such as *gcc*, the ATmega compiler *avr-gcc*, however, is not among them. Note, that to be able to analyze the code Goanna does not need to fully compile the code, it is sufficient to obtain the AST after parsing. Our parser and the ATmega compiler differed in some aspects such as scoping and the use of binary numbers. This required conditioning of the source code, something which could fortunately be achieved by applying a few systematic rewrite rules. A more fundamental problem was that our parser did not retain all of the important information that we needed for the analysis. The most notable example was whether an interrupt service routine has the attribute `SIGNAL` or the attribute `INTERRUPT`; information Goanna requires to perform the Incorrect-Interrupt-Handling check as described in Section 4.1. To implement this check we amended the front-end parser such that this information was available for analysis.

## 5.3 Results

The experiments were performed on a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at 3.4 GHz, 2 MiB L2 cache and 1.5 GiB DDR-2 400 MHz ECC memory. Analyzing all 431 files took 164 seconds, which amounts to an average analysis time of 0.38 seconds, or approximately 1200LoC per second. The maximal runtime was 1.84 seconds. Figure 5 (a) shows the distribution of runtimes. The analysis of all but 2 files (99.5%) took less than 1 second. In 50% of all cases the runtime was smaller than 0.36 seconds.

Figure 5 (b) shows the relation between file size in lines-of-code (after preprocessing) and the runtime. It appears that there is roughly a linear correlation, although there are many outliers. The relation between the number of functions and the analysis time is much more pronounced, as shown in Figure 5 (c). There is approximate a linear correlation with only a few outliers.

At least as important as the runtime are the results of the analysis itself. In the 431 analyzed files, Goanna found 154 errors, i.e., cases that violated one of the



**Fig. 5.** (a) Distribution of the analysis time. (b) Analysis time versus file size in lines-of-code. (c) Analysis time versus file size in number of functions.

three rules defined in Section 4. There were 4 incorrect uses of the timers, 7 cases of incorrect interrupt enabling and disabling, and 143 cases of an assignment to a reserved bit. The examples shown in Section 4 are based on actual code from this code base.

The seven cases of incorrect interrupt enabling all dealt with enabling interrupts in ISRs in which interrupts are disabled by default. These warnings are path dependent, i.e., they could be false positives. However, inspection showed that they are not. The same holds for the warning on incorrect use of timers. They are all legitimate warnings. These 143 warnings do not appear evenly distributed among the projects, but are in six of the 24 projects only. This suggests that some students assign habitually values to reserved registers, while the majority uses them correctly. To the extent that intermediate versions were present, it can be observed that warnings that appeared in earlier versions of the project are also present in the final version. This might be due to the fact that it refers to a compatibility issue that cannot easily be found through testing.

## 6 Conclusion

*Results.* This paper describes why microcontroller C code is different to standard C/C++ code, why tool support has to be particularly tailored and why automatic tools can make a significant contribution to software development in this area.

In particular, we present an extended version of our tool Goanna, which includes static analysis checks specific for microcontroller program code. As such, it is the first of its kind targeted at the ATMEL ATmega16 platform. Since the analysis checks properties that are specific for microcontroller programs such as use of reserved registers, interrupt behavior and timer usage, it is able to detect bugs that otherwise might remain undetected. For the presented case study, Goanna found numerous bugs that had slipped through testing earlier. Moreover, the rate of false alarms was, in this case, zero. This gives a good indication

about the value of integrating Goanna and similar tools into the microcontroller software development process. Since the overhead of running such a tool is negligible, precision is high and real-life bugs can be discovered by non-experts, we estimate a significant decrease in debugging time and in turn in software development time.

We have shown that adding platform-specific checks for microcontroller code to Goanna can be done in a simple and concise manner. This mostly stems from using a model checker as the underlying analysis engine, i.e., that ability to use CTL as a specification language. In contrast, semantic software model checking requires a detailed semantic model, for example, of the memory layout of the specific hardware. While this is possible, as demonstrated by [9], it is a laborious task and run-time behavior does not encourage a compile-time integration of such a tool. On the other hand, semantic analysis allows for a much deeper analysis unearthing even more subtle bugs.

However, Goanna’s abstract rules for interrupt unlocking and locking as shown in Section 4 are similar in nature to specifications used by semantic model checkers. Further investigations are required to quantify the added value semantic software model checkers can provide for finding more bugs and being more precise not only for constructed examples, but for real existing code bases.

*Future Work.* The main challenge to add microcontroller specific checks in this work was to find appropriate CTL formulae, and to deal with a specific non-ANSI dialect of C. Although, the resulting CTL specifications are very succinct, it does take experience to find a specification that does find the right bugs while being low on false positives. To simplify the specification of new properties, we are currently working on a pattern language that allows for the introduction of new specifications without detailed knowledge of CTL.

The other challenge was to deal with the fact that the front-end parser used by Goanna does not natively parse and compile code for the ATMEL ATmega platform. It is an unfortunate fact that different microcontrollers use different dialects of C, which necessitates some rewriting of code and additions to the parser. Fortunately, the kind of analysis that Goanna supports, does not require the ability to compile the code for the target platform.

Future work is to move the application from a lab course within an industrial setting, to a truly industrial setting. This will require to add other platform specific properties and to support other platforms. We have formulated these properties for the ATMEL ATmega16 microcontroller, but they can be easily reformulated for other microcontrollers, provided that we can extend the parser to accept the specific C dialect.

*Opportunities.* Our results suggest that different programmers have different *bug profiles*. This means that tools such as Goanna can be used in software quality assurance to assess the quality and maturity of code, and to some extent also to assess the quality of programmers. Similarly, these tools can also be used in education as a tool to teach platform-specific programming guidelines. Automatic tools provide an immediate feedback, which can be much more effective than

finding and correcting these mistakes through testing or inspection. We considered a code base that has been subject to three different approaches to improve the design and implementation of microcontroller code. The model checking approach followed in [17] puts the emphasis on functional correctness. The work presented in [9] had a focus on the underlying technology and the software development cycle. Our work contributes to these with an automated analysis that improves software quality, maintainability and compatibility across platforms. All three, if not more, are necessary to improve design and implementation of microcontroller applications.

## References

1. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS 2004. LNCS 2988 (2004)
2. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS 2005. LNCS 3440 (2005)
3. Henzinger, T., Jhala, R., Majumdar, R., SUTRE, G.: Software verification with BLAST. In: SPIN2003. LNCS 2648 (2003) 235–239
4. Coverity: Prevent for C and C++. <http://www.coverity.com>
5. Gimpel Software: Flexelint for C/C++ . <http://www.gimpel.com/html/flex.htm>
6. Klocwork: K7. <http://www.klocwork.com/products/klocworkk7.asp>
7. Microsoft: Prefast  
<http://www.microsoft.com/whdc/devtools/tools/PREfast.msp>
8. Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. In: SSV 08. ENTCS 127 (2008)
9. Schlich, B., Kowalewski, S.: Model checking C source code for embedded systems. In: Proc. of the IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation, NASA/CP-2005-212788 (Sept 2005)
10. Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model checking software at compile time. In: TASE 2007, IEEE Computer Society (2007)
11. Schlich, B.: Model Checking of Software for Microcontrollers. Dissertation thesis, RWTH Aachen University (2008)
12. Dams, D., Namjoshi, K.: Orion: High-precision methods for static error analysis of C and C++ programs. Bell Labs Tech. Mem. ITD-04-45263Z, Lucent Technologies (2004)
13. Holzmann, G.: Static source code checking for user-defined properties. In: IDPT 2002, Pasadena, CA, USA (June 2002)
14. Schmidt, D., Steffen, B.: Program analysis as model checking of abstract interpretations. In: SAS '98, Springer (1998)
15. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2. In: CAV 2002. LNCS 2404 (2002)
16. Salewski, F., Wilking, D., Kowalewski, S.: Diverse hardware platforms in embedded systems lab courses: a way to teach the differences. ACM SIGBED Review **2**(4) (2005)
17. Schlich, B., Salewski, F., Kowalewski, S.: Applying model checking to an automotive microcontroller application. In: SIES 2007, IEEE (2007)