Semiformal Verification of Embedded Software in Medical Devices Considering Stringent Hardware Constraints

Lucas Cordeiro[1], Bernd Fischer[1], Huan Chen[2] and Joao Marques-Silva[2]
[1]University of Southampton
$\{lcc08r, b.fischer\}$@*ecs.soton.ac.uk*
[2]University College Dublin
$\{huan.chen, jpms\}$@*ucd.ie*

Abstract

In recent days, the complexity of software has increased significantly in embedded products in such a way that the verification of Embedded Software (ESW) now plays an important role to ensure the product's quality. Embedded systems engineers usually face the problems of verifying properties that have to meet the application's deadline, access the memory region, handle concurrency, and control the hardware registers. This work proposes a semiformal verification approach that combines dynamic and static verification to stress and cover exhaustively the state space of the system. We perform a case study on embedded software used in the medical devices domain. We conclude that the proposed approach improves the coverage and reduces substantially the verification time.

1 Introduction

Embedded systems are ubiquitous in modern day information systems. Our society has become dependent on the services provided by this type of system which consists of a set of hardware/software components that together implement a set of functionalities while satisfying constraints (e.g., timing, power dissipation, and costs). For this kind of system, the choice of the implementation architecture usually determines whether a given functionality is implemented as a hardware or software component.

Due to the high pressure imposed by the market to launch new products coupled with evolving system's specification, semiconductor and system development companies are forced to choose flexible implementations where new products can quickly be built [7]. The increasing computational power and decreasing size and cost of processors, enables system's designers to move increasingly more functionalities to software [20]. Market analysis shows that software-based implementations accounts for more than 80% of system development in embedded systems domain [20].

The increasing number of functionalities being moved to software-based implementations, leads to difficulties in verifying design correctness. In practice, however, this verification is of importance due to dependability properties (reliability and availability) in several embedded system domains such as automotive, industrial automation, and transportation [13]. Nowadays, in order to verify the design correctness of hardware blocks, model checking has been widely used as a verification methodology. Nevertheless, the verification of ESW has always been difficult to be carried out by engineers mainly due to its flexibility.

There is some work that aims to exhaustively generate test vectors, but may require several hours to dynamically verify small ESW using assertion-based verification [14]. Nevertheless, the size of ESW is increasing to millions of LOC and software builds are usually produced on a weekly or daily basis in large organizations. In addition, the use of simulation and assertion-based verification also has limitations to explore the state space and verify more complex properties in ESW. As a result, the approaches adopted to verify design correctness take nowadays 40-70% of the design time to find out bugs and implement the necessary corrections in the design [9].

Despite the vast number of practical applications, embedded system design verification raises a number of hard challenges. There is clearly the need for taking into account stringent constraints (*e.g. real-time, memory allocation, interrupts, and concurrency*) imposed by the hardware when verifying the design correctness of ESW. Hence, this paper proposes a semiformal verification methodology to stress and cover the variables and function calls of the ESW that is under stringent hardware constraints considering microprocessor's Verilog model. Hence, we aim full coverage of the embedded system and reduce the verification time by combining static and dynamic verification.

Outline. Section 2 summarizes related work. Section 3 describes the proposed methodology to verify ESW in medical devices. Section 4 shows the application of model checkers to the verification of the pulse oximeter device. Section 5 shows the experimental results and finally Section 6 concludes this paper and describes the future work.

2 Related Work

This section briefly surveys work that has been published in the area of embedded systems verification. A verification algorithm that uses the compositional backward technique is proposed by [18]. Straunstrup et al. verify six machine models ranging from 10 to 1,421 state machines, but they do not provide information about the characteristics of the embedded software. The authors verify these machine models using generic properties with the *visualState* commercial tool. Straunstrup et al. report a problem to check the properties of a zoom-camera state machine model which contains 36 state machines.

Ivancic et al. present a brief tutorial on model checking of C programs [12]. In this work, the authors describe a verification platform called F-SOFT, which allows to abstract the software modelling and uses customized model checking techniques based on propositional satisfiability (SAT) and binary decision diagram (BDD). Brinksma and Mader present a survey of the basic principles that are involved in the application of model checking to controller verification and synthesis [3], and discuss how model checking can be combined with heuristic cost functions to guide search strategies.

There has been work in the verification of assembly language programs for embedded systems. Thiry and Claesen propose a model checking approach based on BDDs to verify a mouse controller and find inconsistencies between assembly code and flow chart specifications [19]. Balakrishnan and Tahar also propose a similar approach based on the more general multiway decision graph to avoid some BDD-size blow-up [1]. Moreover, Balakrishnan and Tahar also verify a mouse controller and find inconsistencies.

Yun et al. propose a semiformal verification based on PNPM (Petri Net based Representation for Pipeline Modelling) simulation in order to take advantage of both the simulation and the formal verification [21]. Lettnin et al. [15] also describe a semiformal verification methodology that adopts simulation and formal verification. Instead, this solution uses the frontend of BLAST tool [11] to convert a C program to a Control Flow Graph (CFG) and uses SymC model checker [10] to verify the properties. Lettnin et al. apply this methodology in a case study to verify the locking and unlocking rules of a driver.

To the best of our knowledge, there is no work that considers the combination of dynamic and static verification techniques to deal with embedded software written in ANSI-C language using the microprocessor's Verilog model. As a result, our main contribution is a semiformal verification approach to stress and cover variables and function calls of ESW under stringent hardware constraints. As an example, this paper focuses on the medical devices domain. In this sense, we present essentially techniques to formally verify "pure" code and platform specific code at system level with the purpose of exploiting the capabilities of the model checkers using microprocessor models. Simulation tools are also used in order to monitor the variables and function calls in ESW during dynamic verification.

3 A New Semiformal Verification Approach

This section describes our proposed semiformal verification approach to verify ESW. The idea behind this approach is to consider not only higher levels of abstraction, but also the HW/SW interface in platform-based design methodologies [20]. As depicted in Figure 1, our proposed semiformal verification approach starts by designing the unit test for each stage of computation of the system's functionalities that are described in the product backlog [2]. The product backlog may be viewed as an evolving, prioritizing queue of requirements to be developed in an iterative and incremental way [16]. Therefore, according to the business value (priority) of the system's functionalities defined in the product backlog, we must first write the unit test for a given functionality and thereafter we must compile successfully the unit test before really writing the functionality's code.

There are some benefits if we design and compile the unit test before writing the code [7]. These benefits fall into the following categories: *(i)* reflect about the design before coding the system functions, *(ii)* test the correctness of the functionalities by stressing and covering difficult scenarios, and *(iii)* achieve better results in terms of cyclomatic complexity v(ϕ) in order to facilitate the verification activities. Low cyclomatic complexity v(ϕ) levels (where ϕ means the CFG of the program) make white-box testing easier due to the fact that they decrease substantially the number of paths (i.e., the control flow) that should be tested to reasonably guard against errors [7].

After designing the tests and developing the ESW, the model translator converts the ESW and microprocessor model to BDDs (Binary Decision Diagrams) or CNF (Conjunctive Normal Form) formulas with the purpose of allowing the engineer to verify that certain system's properties hold in the model by using BDD-based or SAT-based model checking techniques. The system's properties (e.g., deadlock-freedom, reachability, safety and liveness) can be expressed as state formulae using Computation Tree Logic (CTL), Real-Time CTL, Linear Temporal Logic (LTL), and Property Specification Language (PSL) [4].

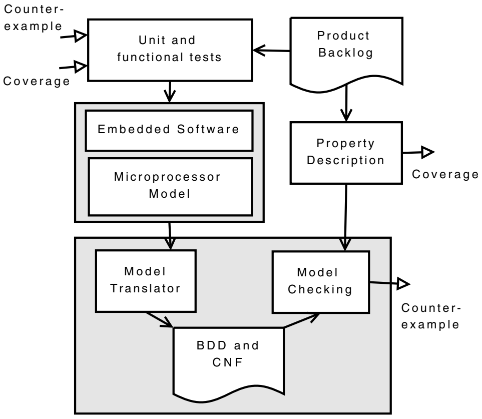

Figure 1. Semiformal Verification Approach.

Furthermore, if the model does not satisfy the specification then a counter-example is automatically generated. This counter-example is included into the test suite and can be used to test the ESW during dynamic verification. The process to convert the counter-example to test case is still performed manually as described in Section 4. Another important aspect is that the verification directive *cover* of PSL can be included in order to measure the occurrence of expected behaviours during dynamic verification. However, we did not integrate yet tools in our proposed approach that provide information about property coverage.

In a complex embedded system, there will be several software modules that depend on the hardware and others that do not. We usually cannot determine whether a problem is within a module or is somewhere else in the system when we integrate all components into the embedded platform. This often happens in embedded systems due to timing constraints. Specially in embedded hard real-time systems (e.g., pulse oximeter), where deadlines should not be missed, it is extremely important to reason quantitatively about temporal properties to assure the correctness of the design.

Hence, when we try running newly developed ESW on a microprocessor model, we are tackling many unknowns simultaneously. A problem on the simulator, CPU model, register address access, or the interruptions generation can mask as a software bug leading to a significant and frustrating waste of time. Specially, when we flash our ESW into the platform, hardware that worked perfectly one minute can be buggy the next and intermittent hardware bugs are hard to deal with. Consequently, we still need an efficient approach to isolate completely the modules in order to avoid verifying hardware and software simultaneously. As a result, we propose three approaches to tackle these problems. In the first approach, we aim to verify statically and dynamically the "pure" ESW while the second approach focuses on the hardware-dependent code. The third approach considers the integration of hardware and software components so that it allows us to find properties violations related to system integration during the whole lifecycle of the system development.

3.1 1st Approach: Verification using Platform-Independent Software

In order to avoid verifying hardware and software simultaneously, we have to implement small changes in the ESW to keep the capability of *(i)* using model checkers, *(ii)* performing automated unit tests, and *(iii)* running the ESW on the target platform. Furthermore, these modifications are needed because the model checkers (e.g., CBMC [5] and SATABS [6]) and the unit test framework (e.g., EmbUnit[1]) that accept ANSI C do not recognize platform-dependent software and so we have to abstract this information from our C models. The second approach described in the next subsection provides techniques to verify formally the platform-dependent software using microprocessor's Verilog model with the model checker CBMC [5].

In order to avoid the need for significant usage of the `#ifdef` construct, we include the platform-dependent software in lower level driver files with the purpose of isolating platform-independent and platform-dependent software. This then allows the engineer to efficiently verify the "pure" software and easily migrate it for each target platform that we intend to reuse the ESW on. This approach also helps understand clearly the interface between "pure" code and hardware specific code. For the software modules that are hardware specific, we also separate into two different classes: modules that *control the hardware* and modules that are *driven by the environment*. For the latter, the strategy is to use stubs in order to simulate the interaction between the environment and the system. For instance, we could collect actual data from the pulse oximeter sensor hardware in real-time and afterwards put them into a file that can be used by the ESW during the simulation phase (i.e. dynamic verification).

As a result, we can adapt our function to read the data from the file stored on the PC instead of reading this information directly from the sensor. This type of strategy is applicable to a wide range of medical devices, because most of them have a data acquisition stage, then the application of an algorithm, followed by output of a result. In addition to that, the actual data collection is usually feasible because the manufacturer of the sensor often provides an evaluation board that connects it to the PC and then allows the engineer to test the sensor hardware independently from the target platform. The main benefit of this approach is that

[1] embUnit: Unit Test Framework for Embedded C Systems. http://embunit.sourceforge.net/

it allows the embedded system engineer to isolate problems that might be related to the microprocessor model. Thus, by gaining confidence in the ESW through static and dynamic verification, we can then verify the ESW knowing that the only areas left, are direct interaction with hardware.

3.2 2nd Approach: Verification using Platform-Dependent Software

In order to verify platform-dependent software, we adopt verification techniques based on assertions to reduce substantially the problems related to HW/SW interaction. Hence, C's `assert` macro can be used to state an assumption (e.g., `assert(next<buffer_size)`) so that execution will halt if the asserted property does not hold. This halt allows the engineer to examine the call stack (during dynamic verification) or the counter-example (during static verification) and check what went wrong. The following C code shows an example of hardware-dependent software of the pulse oximeter timer:

```
1 oc8051_tc.th0=THIGH;
2 oc8051_tc.tl0=TLOW;
3 for(cycle=0; cycle<n; cycle++)
4   next_timeframe();
5 assert(oc8051_tc.th0==X);
6 assert(oc8051_tc.tl0==Y);
```

The registers `th0` and `tl0` are loaded with the values THIGH and TLOW in lines 1 and 2 respectively. These values define the time to generate an interrupt in the system that is then used to trigger some event. The function `next_timeframe()` in line 4 changes the state of the register in the Verilog model *n* times according to the loop defined in line 3. After that, the content of the registers are then checked in lines 5 and 6 using `assert` macros. Consequently, the `assert` macro can be used in the presence of a microprocessor model in order to verify the interplay of HW/SW components. For this purpose, the CBMC model checker allows the engineer to reason about a C/C++ program together with a Verilog model of the hardware. Given this, this model checker provides means to *(i)* change the state of the Verilog model from the C program, *(ii)* synchronize inputs between the software and hardware modules, *(iii)* restrict the choice of the Verilog module inputs from the C program, and *(iv)* map variables within the module hierarchy in order to monitor the registers values during static verification.

The use of the `assert` macro is appropriate during system development, but the code should not halt when it is in customer use. In order to tackle this problem, we use a wrapper function that allows to use the plain `assert` macro or quietly write a diagnostic message to a buffer, depending on a flag that indicates development or field environment. Each message written to the buffer reports the source file, line number, severity, and diagnostic text. Although the use of the assertion-based verification is quite efficient, it has limitations to verify more complex properties in ESW.

In this context, CBMC model checker is only capable of verifying more complex properties that arise from the nontrivial interplay of HW/SW components if we implement the complete specification of the property in ANSI-C. However, the combination of Real-Time CTL and PSL is more effective by the fact that it allows the verification of complex temporal and functional properties and can create functional coverage models built on formally specified properties. As future work, we intend to define a subset of Real-Time CTL and PSL to verify platform-dependent software in CBMC in order to improve the coverage of the system.

3.3 3rd Approach: Domain-Level Verification

The specification languages such as FLTL (Finite Linear Temporal Logic) and Real-Time CTL seem to be a suitable vehicle for specifying properties that involve explicit time bounds. However, even if we use the model checker NuSMV2, it is still premature to consider the verification of temporal properties in ESW, because it considers that each transition takes unit time for execution. For embedded systems, we need mechanisms to assign values to each transition so that these values are the estimated worst-case execution times (WCET) of the respective transition on the selected processor. Given this, we clearly need a domain-level verification strategy in order to isolate problems observed at the system level, such as timing delays. Figure 2 shows our proposed domain-level verification process.

This process allows the team members to integrate components into the system, manage the product development line, and hence identify problems at system level. This process works as follows: *(i)* firstly we must check out the code in the repository to a local workspace. Thist allows us to implement new system's functionalities, fix bugs and improve the system's code. Moreover, we are also able to generate new product builds in the local repository. The *Subversion* (SVN) repository has the purpose of controlling the system's code versions.[2] *(ii)* Before making the ESW available in the repository to other engineers, we first specify and verify formally the system's properties using the microprocessor's Verilog model (static verification) and then run all test cases in an automated way.

If the model checkers find bugs or violation of properties during the verification process, then the counter-examples are converted into test cases and included into the EmbUnit

[2] SVN (http://sourceforge.net/svn/) is an improved version of the well-know CVS (Concurrent Versions System) available at http://sourceforge.net/cvs/

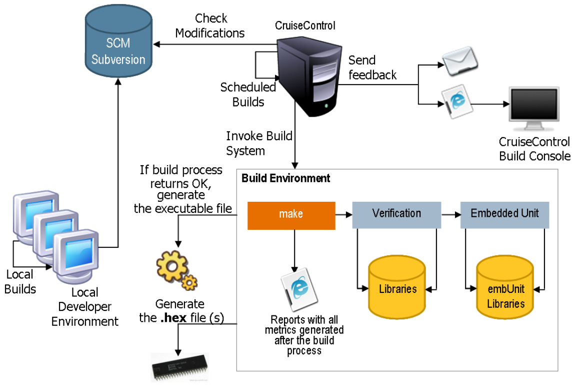

Figure 2. Domain-Level Verification Process.

tool. This tool aims to perform all kinds of unit, functional, and integration tests. *(iii)* After making the ESW available in the SVN repository, the *CruiseControl*[3] tool looks for code modifications in the repository. If the file date/time changes then the *CruiseControl* tool starts the build process in an automated way. If there is a compilation error then *CruiseControl* tool sends an e-mail to the person responsible for breaking the code. Otherwise, it generates the .hex file that will be loaded into the flash memory of the embedded platform and it then runs other tools (e.g., *CCCC*[4], CBMC, Satabs, NuSMV2, and *EmbUnit*) in order to capture the metrics, verify and test the code. Data on source code size, number of functions and cyclomatic complexity are obtained using CCCC tool which analyzes C/C++ files.

3.4 Comparison of Proposed Approaches

As mentioned in the previous subsection, we need a domain-level verification strategy in order to isolate problems observed at the system level, such as timing delays. Given this, our first approach focuses on the platform independent software and allows to determine quickly property violation in the control and data flow of the ESW. This approach essentially abstracts the platform details on the cost of becoming hard to find timing delays when all system components are integrated into the embedded platform. The second approach aims to verify formally the HW/SW interaction and find problems related to ESW that controls the hardware components. On the other hand, this approach does not consider higher levels of ESW as the first approach does. The third approach then aims to integrate the HW/SW components and perform static and dynamic verification to find properties violations that might arise due to the integration of the system's components.

4 Medical Device Case Study

This section describes the main characteristics of the pulse oximeter equipment and shows the application of the model checkers CBMC, SATABS, and NuSMV2 to the static verification of the functional and temporal properties of the pulse oximeter. Generally speaking, the pulse oximeter is responsible for measuring the oxygen saturation (SpO_2) and heart rate (HR) in the blood system using a non-invasive method [7]. The pulse oximeter that we used to apply our semiformal verification approach, also allows the user to change the sample time and save SpO_2 and HR values in the device's memory in order to transfer them to the PC desktop for further analysis.

The system architecture is composed of the application software, platform API and architecture. The hardware solution consists basically of two platforms: a data *acquisition platform* that is responsible for providing the SpO_2 and HR levels, and a *development platform* that aims to capture the data provided by the acquisition platform, perform a set of computations and finally show the results on a LCD display to the user. The development platform is based on the AT89S8252 which has an 8051-like architecture with code and data memory integrated on chip. The software architecture is composed, basically, of device drivers (i.e., display, keyboard, serial, sensor, and timer) that are hardware-dependent code, a system log component that allows the developer to debug the code through data stored on RAM memory, and an API that enables the application layer to call the services provided by the platform.

In order to verify dynamically the pulse oximeter ESW, we use the Keil μVision IDE and Debugger[5] as well as the EmbUnit tool. Keil μVision provides an interrupt system, four 8-bit parallel ports for taking inputs and presenting outputs, one serial channel, and three timers that can be used as counters. There is also a performance analyzer that helps visualizing the time `(max, min, avg)` spent in executing each function of the pulse oximeter. EmbUnit tool provides means to apply assertion-based verification in ESW written in ANSI C. It provides a set of macros to assert strings, integers, pointer value, conditions, and to register a failed assertion with the specified message.

4.1 Formal Verification using Model Checking

In order to show the suitability of each model checker to verify formally ESW, we present here the advantages and disadvantages of applying CBMC, SATABS, and NuSMV2 to the verification of four properties that we extracted from the pulse oximeter product backlog. For the first property *(a)*, we intend to check if the tail of the circular buffer

[3] CruiseControl. http://cruisecontrol.sourceforge.net/

[4] C and C++ Code Counter. http://sourceforge.net/projects/cccc

[5] Keil software. http://www.keil.com/

```
1 void insertLogElement(Data8 b) {
2     if (next < buffer_size) {
3           buffer[next] = b;
4           next = (next+1)%buffer_size;
5           assert(next<buffer_size);
6     }
7 }
```

Figure 3. Function to insert an element into the circular buffer.

points to its head and in the second property *(b)*, we intend to check if fixed-length text messages are included and removed from the circular buffer using the FIFO (First In, First Out) policy. These two properties belong to the log component of the pulse oximeter. For the third property *(c)* that belongs to the sensor driver, we aim to verify the data flow to compute the HR value that is provided by the pulse oximeter sensor hardware. For the fourth property *(d)* that belongs to application software, we intend to verify if the user is capable of adjusting the sample time of the embedded device.

4.1.1 CBMC and SATABS

CBMC is capable of handling the full ANSI C language using Bounded Model Checking (BMC) [5]. To check the property *(a)*, we need to include the `assert` macro (line 5) as shown in Figure 3. CBMC was not capable of verifying property *(a)*. To verify this property in CBMC, we need to initialize the *first, next, buffer_size* variables to proper values. As CBMC is used to verify automatically array bounds and pointer safety as well as `assert` macros specified by the engineer, then we have to develop some additional code and use the `assert` macro in order to verify the property *(b)*.

It is important to mention that we could not verify the properties *(c)* and *(d)*, because CBMC does not support any mechanism to specify more complex properties. However, we can overcome these problems by expressing properties *(c)* and *(d)* using LTL and translate them to $Büchi$ Automata using Wring tool [17]. After that, we convert each $Büchi$ Automata representing the property into ANSI-C and merge them into the code. The properties *(c)* and *(d)* can be expressed using the following LTL pattern:

$$AG\,(p \rightarrow Fr) \qquad (1)$$

As an example, for the property *(c)*, let *p* denote the state that the buffer contains HR and SpO_2 raw data. Let *r* denote the state that defines the respective HR value. Consequently, any state containing the HR and SpO_2 raw data in the buffer is eventually followed by a state representing the respective HR value. The same approach is applied to verify property *(d)*. Furthermore, CBMC was able to identify bugs related to array bounds and pointer safety in functions that implement these two properties. An important aspect of CBMC is the capability of verifying hardware-dependent software. CBMC supports co-verification and allows us to verify our ANSI-C code together with a Verilog model of the hardware. Thus, CBMC is able to cope with ESW that interacts directly with hardware modules. As with CBMC, SATABS can also verify automatically array bounds, pointer safety, exceptions, and `assert` macros specified by the engineer [6]. As a result, we faced the same scenario as the CBMC tool in order to verify all four properties *(a), (b), (c),* and *(d)*. Actually, we identified much more bugs in our ESW by using SATABS than with the CBMC tool. Moreover, there were bugs that were discovered only with CBMC and others only with SATABS.

```
MODULE log
  VAR
        buffer_size : 0..255;
        nextptr : 0..255;
  DEFINE
        nextptr_condition := nextptr < buffer_size;
  ASSIGN
        init(nextptr) := 0;
        next(nextptr) := case
        nextptr = nextptr_condition & buffer_size > 0
                    : ((nextptr+1) mod buffer_size);
        1 : nextptr;
        esac;
PSLSPEC AG nextptr ≤ buffer_size;
```

Figure 4. Function insertLogElement in NuSMV language.

4.1.2 NuSMV2

NuSMV2 is a symbolic model checker that combines BDD-based and SAT-based model checking [4]. NuSMV2 accepts system models written in NuSMV language which can represent synchronous and asynchronous FSMs. The system properties can be expressed in CTL, Real-Time CTL, LTL and PSL. Hence, in order to verify property *(a)*, we had to convert the ANSI-C code from Figure 3 to NuSMV language. Figure 4 shows the abstracted version of the *insertLogElement* function in NuSMV language.

The MODULE log is split into four different sections: VAR, DEFINE, ASSIGN, and PSLSPEC. The VAR section declares the state variables of the module and their ranges (note that the AT89S8252 is a 8-bit processor). The DEFINE section defines the symbol *nextptr_condition* that incorporates the expression *nextptr* $<$ *buffer_size* in order to make the description more concise. The ASSIGN section contains the FSM responsible for controlling the circular buffer operations. As can be seen in line 14, we use PSL to specify the property *(a)*. The property *AG nextptr* $\leq$ *buffer_size* ensures that on all paths, at all states on each path the formula *nextptr* $\leq$ *buffer_size* is satisfied. When we verified the property *(a)* in NuSMV2, we found a property violation that was not identified by CBMC and SATABS. NuSMV2 identified a *division by zero* in line 11 (and then we had to adjust our model and include the condition *buffer_size* $>$ *0* in line 10).

Another important point is that CBMC and SATABS are not capable of identifying properties violations related to data types. For instance, if we declare the *nextptr* as *int* and *buffer_size* as *unsigned int*, then the *nextptr* will range from -127 to 127 and *buffer_size* will range from 0 to 255. In this situation, the *nextptr* will never reach the *buffer_size*, if the latter is set to 128, for instance. Fortunately, NuSMV2 is capable of identifying this sort of property violation by representing data types as scalars and defining its range. However, NuSMV2 is not capable of verifying property *(b)* due to the fact that it involves dataflow and for each operation we need some mechanism to save the value of the variables. Nonetheless, mere unit test can be used to verify property *(b)*. For properties *(c)* and *(d)*, we verified them formally by using CTL and PSL in combination. As an example, the property *(d)* can be expressed using the following PSL formula:

$$PSLSPEC\ always\ (p \wedge q \longrightarrow eventually!\ r) \qquad (2)$$

In order to check if the sample time is incremented, let *p* denote the state that the sample time is selected and let *q* denote the state that the user pressed the button to increment the sample time. Let *r* denote the state that the sample time is incremented. Consequently, the PSL formula checks if any state satisfying $p \wedge q$ is eventually followed by a state satisfying *r*.

5 Experimental Results

This section describes the experimental results when applying the proposed approach in the verification of the pulse oximeter equipment. These experiments were conducted on an Intel Core 2 Duo CPU, 2Ghz, 3GB RAM with Linux OS. The final version of the pulse oximeter ESW has approximately 3500 lines of ANSI-C code and 80 functions. In order to meet the application's deadline, there are 100 lines of Assembly code that are responsible for writing text messages to the LCD hardware. By applying the proposed approach, we found 14 violations of the verification conditions (VCs) using SATABS, 6 violations of the VCs using CBMC and 3 properties violations using NuSMV2. The verification conditions are related to array bounds and pointer safety and are automatically generated by the model checkers CBMC and SATABS.

Table 1 shows the total number of verification conditions (#C), number of properties and test cases as well as the verification time (T) in seconds for each module of the pulse oximeter ESW using the tools CBMC, SATABS, NuSMV2, and EmbUnit. We achieved short time during dynamic verification due to the fact that our first approach (described in Section 3.1) separates logic from timing while running the ESW against the EmbUnit tool on the PC desktop. On the other hand, in order to verify dynamically the timing constraints at system level, we run the hardware unit tests and monitor the register values in the Keil μVision tool. SATABS verified more VCs and NuSMV2 verified more properties, but these model checkers did not take into account the hardware-dependent code. We use an assertion-based verification with CBMC in order to verify the hardware-dependent code. We were able to specify more properties of the pulse oximeter with NuSMV2 due to the fact that it supports a wide range of specification languages and then some properties are checked in more than one specification language.

6 Conclusions and Future Work

In a complex embedded system there will be several software modules that depend on the hardware and others that do not. Our semiformal verification approach allows the engineer to reason quantitatively about functional and temporal properties to assure the timeliness and correctness of the design. Hence, we propose a combination of techniques to verify statically and dynamically the "pure" and hardware-related ESW as well as techniques that aim to find properties violations at system level. Apart from the state space explosion problem, CBMC and SATABS allow us to verify full ANSI-C, but these model checkers have limitations to specify more complex temporal properties in ESW. NuSMV2 provides a variety of languages to specify the system's properties, but there is no straightforward mapping from ANSI-C to NuSMV language. In the near future, we intend to verify formally ANSI-C and SystemVerilog models using the Satisfiability Modulo Theories (SMT) solver Z3 [8] in order to make use of high-level information and then reduce substantially the state space to be explored. Furthermore, we aim at defining a subset of Real-Time CTL and PSL to verify more complex properties in embedded

	Static Verification								Dynamic Verification	
	CBMC			SATABS			NuSMV2		EmbUnit	
Module	Properties	#C	T(s)	Properties	#C	T(s)	Properties	T(s)	Test Cases	T(μs)
MenuApp	9	32	5	9	23	3	53	5.4	62	130
Sensor	10	224	20	10	617	130	10	3.7	42	403
LCD	1	22	0.02	1	1	0.02	8	4.1	6	6
Serial	6	5	2	1	1	0.02	8	4.6	5	8
Timer	11	12	4	3	13	2	7	4.9	7	12
Keyboard	1	1	0.02	1	1	0.02	16	4.8	10	18
Log	4	14	2	4	6	1	4	5.4	10	48
Total	**42**	**310**	**33.04**	**29**	**662**	**134.06**	**106**	**32.9**	**139**	**625**

Table 1. Results of the proposed approach.

systems using SMT-based CBMC approach instead of SAT solver.

Acknowledgement We thank Daniel Kroening for many helpful discussions about CBMC and SATABS model checking tools.

References

[1] S. Balakrishnan and S. Tahar. On the formal verification of embedded systems using multiway decision graphs. *Technical Report TR-402, Concordia University, Montreal, Canada*, 1997.

[2] K. Beck and C. Andres. *Extreme Programming Explained - Embrace Change*. Second Edition, Addison-Wesley, 1999.

[3] E. Brinksma and A. Mader. Model checking embedded system designs. *Proceedings of the Sixth International Workshop on Discrete Event Systems (WODES'02). ISBN 0-7695-1683-1*, page 151, 2002.

[4] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV: a new symbolic model checker*. http://nusmv.itc.it/, 2009.

[5] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 168–176, 2004.

[6] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *LNCS*, pages 570–574, 2005.

[7] L. C. Cordeiro, R. S. Barreto, R. F. Barcelos, M. Oliveira, V. F. Lucena Jr., and P. Maciel. Txm: An agile hw/sw development methodology for building medical devices. *In ACM SIGSOFT Software Engineering Notes. ISSN:0163-5948*, 32(6):32, 2007.

[8] L. de Moura and N. Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[9] H. Goldstein. Checking the play in plug-and-play. *Spectrum, IEEE*, 39(6):50–55, 2002.

[10] F. M. Group. *SymC*. http://www-ti.informatik.uni-tuebingen.de/ fmg/symc/, 2008.

[11] T. A. Henzinger, D. Beyer, R. Majumdar, and R. Jhala. *BLAST: Berkeley Lazy Abstraction Software Verification Tool*. http://mtc.epfl.ch/software-tools/blast/, 2009.

[12] F. Ivancic, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking c programs using f-soft. *Computer Design: VLSI in Computers and Processors, 2005. ICCD 2005*, pages 297–308.

[13] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 2002.

[14] D. Lettnin, P. K. Nalla, J. Ruf, , T. Kropf, and W. Rosenstiel. Verification of temporal properties in automotive embedded software. *Design, Automation and Test in Europe*, pages 164–169, 2008.

[15] D. Lettnin, P. K. Nalla, J. Ruf, R. Weiss, A. Braun, J. Gerlach, T. Kropf, and W. Rosenstiel. Semiformal verification of temporal properties in embedded software. *GI/ITG/GMM Workshop, Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Erlangen, Germany*, 2007.

[16] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. First Edition, Series in Agile Software Development, Prentice Hall, 2002.

[17] F. Somenzi and R. Bloem. Efficient buechi automata from ltl formulae. In *CAV 2000, LNCS 1855:247263*. Springer-Verlag, 2000.

[18] J. Straunstrup, H. Andersen, H. Hulgaard, J. Lind-Nielsen, G. Behrmann, K. Kristoffersen, A. Skou, H. Leerberg, and N. Theilgaard. Practical verification of embedded software. *IEEE Computer*, 33(5):68–75, 2000.

[19] O. Thiry and L. J. M. Claesen. A formal verification technique for embedded software. *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors. ISBN:0-8186-7554-3*, pages 352–357, 1996.

[20] A. S. Vicentelli, P. L. Carloni, F. Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. *Proceedings of the Design Automation Conference*, (41):409–414, 2004.

[21] Z. Yun, L. Xi, Z. Siyang, and G. Yuchang. Implementation of a semi-formal verification for embedded systems. *Proceedings of the Second International Conference on Embedded Software and Systems (ICESS'05). ISBN: 0-7695-2512-1*, page 7, 2005.