**Proceedings of the ASME 2011 International Design Engineering Technical Conferences &
Computers and Information in Engineering Conference
IDETC/CIE 2011
August 28-31, 2011, Washington, DC, USA**

DETC2011/MESA-47139

HARDWARE SUPPORT FOR EFFICIENT TESTING OF EMBEDDED SOFTWARE

Thomas Reinbacher
Andreas Steininger
ECS Group
Inst. of Computer Engineering
Vienna University of Technology
Treitlstr. 3, 1040 Vienna, AT
lastname@ecs.tuwien.ac.at

Tobias Müller
Martin Horauer
Dept. of Embedded Systems
Univ. of Applied Sciences Technikum Wien
Höchstädtpl. 5, 1200 Vienna, AT
tobias.mueller@technikum-wien.at
horauer@technikum-wien.at

Jörg Brauer
Stefan Kowalewski
Embedded Software Laboratory
RWTH Aachen University
Ahornstr. 55, 52074 Aachen, DE
lastname@embedded.rwth-aachen.de

ABSTRACT

Verification of software for embedded systems is crucial for ensuring a product's integrity. Formal approaches like static analysis and model checking are gaining momentum in this context. To make an exhaustive examination of the system's state space tractable in practice, these methods perform an abstraction and over-approximation of the possible behavior. As a side-effect, however, this leads to "false negatives" – property violations that exist only in the model and not on the real system. Ruling out such spurious property violations by manual valuation is a tedious and error-prone process. This paper reports on the concepts and design of a hardware unit to support the identification of false negatives. Our approach has several advantages: (i) It works on microcontroller binary code, thus avoiding the need for availability of high-level source code, and covering compiler bugs as well. (ii) Moving the verification directly to the target platform rules out modeling errors. (iii) The cases suspected to lead to spurious property violations can serve as very efficient test cases for a specific implementation later on. We illustrate principle and benefits of the proposed approach by a worked example.

1 INTRODUCTION

Today, computer-based systems are entrusted safety-critical functions in many application domains, including, but not limited to, automotive, avionics or aeronautics. In such applications, the failure of the computer system may have (by definition) catastrophic consequences in terms of loss of human lives, severe environmental damage, and/or huge cost. Therefore, it is – among other provisions – mandatory to thoroughly test these systems in order to unveil residual design and implementation/fabrication bugs before they are put into operation. While the structural, scan-based test employed for hardware scales reasonably well with complexity, the traditional methods for testing embedded software like code reviews and static tests (e.g., checks for memory leaks, coding rules, runtime analysis) alone are not adequate for the complexity of contemporary applications. It is thus desirable to derive test data automatically.

1.1 Primer on Formal Verification

In recent years, technical contributions have advanced formal methods to the point they can be applied to automatically verify (industrial-sized) software against a user-supplied specification. In contrast to the traditional, test case driven approaches with limited coverage, formal verification techniques — in principle — exhaustively check all paths through the program against its requirements (or specification). Most notable formal methods are (i) deductive methods such as theorem proving, (ii) symbolic or explicit exploration of state spaces by model checking [1], and (iii) analyzing abstract executions of programs by means of abstract interpretation [2]. With a practical application in mind, all of them carry their own strengths and weaknesses.

Deductive methods are very elegant, but often considered expensive, since they rely on experts in the field who guide the verification process by providing additional hints to a theorem prover or integrating manually derived proofs [3]. Model checking involves an exhaustive exploration of all states and transitions of (a model of) the system-under-test. If the model checker unveils a bug, it does so by providing a counterexample which exhibits a path to the failure. Counterexamples are thus extremely valuable for debugging [4]. A notorious difficulty with model checking is that state explosion occurs often already in moderately complex applications. Because of this complexity, there has been much interest in representing states of a program symbolically, which enables states that share some commonality to be represented without duplicating their commonality.

By way of comparison, the key idea in abstract interpretation is to abstract from the detailed nature of states. A program analyzer then operates over collections of states, which are equivalent in some sense, rather than individual states. If the number of collections of states — called abstract states — is small, then all paths through the program can be checked without incurring the problem of state explosion. This approach can be seen as an automatic simplification of the system model in order to enable verification. However, when too many details are lost when working with abstract states the approach cannot prove the correctness of a system; the technique leads to *false negative* warnings. Furthermore, deriving counterexamples from abstract interpretation is difficult [5]. The cumbersome task of inspecting and eliminating spurious error reports resulting from abstract interpretation thus remains with the user.

Pure testing, on the other hand, does not guarantee complete coverage of the reachable state space of the program, but does not produce spurious property violations. Typical coverage criteria have, however, limited value [6] for deriving test cases to unveil bugs. It is thus a challenging task to derive test cases which either exhibit a bug or show its absence.

1.2 Verification-driven test-case generation

On the contrary, we present a method which combines testing with abstract interpretation so as to overcome the problems inherent to either method. The key idea of our approach called CEVTES is to use abstract interpretation in order to detect *potential* violations of the specification. Backward analysis is then applied to derive a set of *potential* traces, which could describe a path to the defect in the concrete program. Since this set is derived from an abstraction of the program, it might contain spurious traces, which cannot occur in the concrete program. We thus execute the test trace on the target hardware in order to filter spurious traces. To do so, we have developed a dedicated hardware unit, which allows to check relevant properties on the fly, while the program executes on a microcontroller IP core. Overall, this approach confers the following advantages:

1. Checking user-defined invariants directly in the system execution yields an end-to-end verification that covers compiler bugs [7], modeling faults (including processor specification faults) and similar issues that are not addressed by other formal, model-based approaches.
2. While the original intention of the checking unit is to support the software verification process, it can conveniently be used for on-line error detection during operation as well. Such a plausibility checking, addressing both, control flow as well as data ranges, is a very effective support for fault tolerance.

1.3 Structure

The paper is structured as follows: Section 2 briefly surveys related work, followed by Section 3 which details certain aspects of our test-case generation approach. Then, we formalize our specification language in Section 4. Subsequently, in Section 5 the property monitoring unit is presented, for which Section 6 presents some implementation results. Finally, Section 7 concludes the paper.

2 RELATED WORK

Runtime verification [8,9] is an akin research area, combining formal verification and program execution. Runtime verification monitors certain requirements (such as, e.g., an LTL formula with adaptation to finite traces) during the normal operation of the system. While classic runtime verification can only increase the confidence in the program under scrutiny by checking that the current execution is consistent with the expected behavior, our approach actively drives test case generation towards finding real-life bugs. Nevertheless, our work allows property monitoring in a stand-alone fashion even without the presence of a test-case generation framework.

Classical hardware monitors that simply probe one or more internal signals have been known in literature for a few decades, such as the non-interference monitoring and replay mechanism described by Tsai et al. [10]. While their intention is to replay an execution from an execution history our hardware property unit sharpens abstract interpretation. Drechsler [11] describes an approach to synthesize checkers for online verification of SoC designs, but does not allow for checking arithmetic relations among bit-vectors.

Mercer and Jones [12] combine a cycle accurate debugger with model checking algorithms to derive a model of execution at machine-code level. Their work is based on the GNU debugger (`gdb`) with support for different processor backends. Their approach is only feasible for a small sized code base and suffers from state explosion.

3 THE CEVTES APPROACH

CEVTES[1] is an embedded systems testing framework [13]. It uses abstract interpretation of microcontroller binary code to generate assertion-directed test cases. The specification and the resulting test cases are transfered to a hardware monitor unit that decides validity of warnings. CEVTES currently supports the Intel MCS-51 architecture and is easily adaptable to other target architectures.

Our framework as outlined in Fig. 1 takes an executable binary file and a specification as inputs. The binary file is the unmodified object code generated by the compiler from an arbitrary high-level programming language. The specification may contain local assertions and global invariants, as described in Sect. 4. After input parsing, we generate an initial control flow graph (CFG) of the binary code and apply abstract interpretation to derive program invariants. These invariants are used by the test-case generator to identify possible specification violations. Next, a backward analysis derives actual program inputs that lead the execution to the possible specification violation. The test traces are then transferred to and executed on real hardware, i.e., an IP-core instance of the target microcontroller running within an FPGA embedded in its operating environment. A property monitoring unit (PMU) attached to the IP core tracks specification items during test-case execution and provides runtime feedback, as will be shown in Sect. 5.

3.1 Preliminaries

In the following, we describe notations used in the remainder of the paper.

State of a microcontroller program: Let a state of a program be a tuple $\langle pc, m\rangle \in \mathsf{Locs} \times \mathsf{Mem}$, where Locs is a finite set of program locations, and Mem represents the set of all possible memory configurations of the microcontroller. Then, the state space of the program is a subset of $\mathsf{Locs} \times \mathsf{Mem}$. A memory location may either be a register r or a special function register sfr. For the Intel MCS-51 architecture, in order to address a specific register in the internal RAM area, we write $r_x : \{x \in \mathbb{N}_0 \,:\, 0 \leq x < |\mathsf{Mem}|\}$. For example, r_{10} denotes the memory location at address 10. Special function registers are referred to by an unambiguous string representation, such as ACC for the accumulator.

Local assertion: A *local* assertion $\mathscr{A}(pc, \varphi)$ is a property φ attached to a certain program location $pc \in \mathsf{Locs}$.

Global invariant: A *global* invariant $\mathscr{I}(\varphi)$ is a property φ intended to hold for all program locations Locs in every execution.

[1]http://ti.tuwien.ac.at/ecs/research/projects/cevtes

3.2 Test-case generation

We use abstract interpretation to generate property-directed test cases. The remainder of this section gives a brief introduction, the interested reader is referred to [13, 14] for further details.

Abstract interpretation: Abstract interpretation automatically infers dynamic properties of programs, see [2]. The key idea is to simulate the execution of each concrete operation $g : C \to C$ (in our case a single microcontroller instruction) using an abstract analogue $f : D \to D$, where C and D denote the domains of concrete values and descriptions. Each abstract operation f is designated to model its concrete counterpart g in the following sense: If $d \in D$ describes a concrete value $c \in C$, then the result of applying g to c is described by applying f to d. Representatives of abstract domains D are the non-relational interval domain [15] and the relational variants thereof.

Analysis: To derive a set of test cases, our abstract interpretation framework derives invariants using interval analysis and synthesized transformers for basic blocks. In detail, we synthesize optimal transfer functions from propositional encodings of the microcontroller instructions' semantics using SAT solving. In case the invariants exhibit a potential property violation, a backward analysis derives a path (the test case) from the property violation to the start of the program. This is done by the test-case generator.

3.3 Test-case format

The test-case generator (cf. Fig. 1) derives a test suite Γ. Γ is a finite set of n test cases t, each of which is a quintuple $t = \langle P, \pi, \Theta, E, O\rangle$ with:

P : Binary program code. The `*.hex` file, which is equivalent to the user-supplied binary file, is loaded into the microcontroller's PROM. Note that we neither alter the source code nor do we insert additional event-triggers into the program, which is common practice in runtime verification.

π : Control flow information. A finite path representing the actual test trace by a sequence $\langle pc_0, \ldots, pc_n\rangle$ of program counter locations. The sequence is the predicted execution history (temporally ordered pc locations) of t.

Θ : Specification items. A list of global invariants $\mathscr{I}(\varphi)$ that need to be tracked by the monitor unit. Local assertions, though also considered part of the test case, are not listed here, as they need not be transferred to the hardware.

E : Environment information. A list of external inputs $In := \langle pc, i\rangle$, given by an input i and corresponding program locations pc where the input should be applied to the microcontroller. Since embedded applications heavily interact with the environment E is vital to drive execution towards the property violation.

O : Execution options. Additional settings to be applied to the IP-core instance, such as the system clock speed.

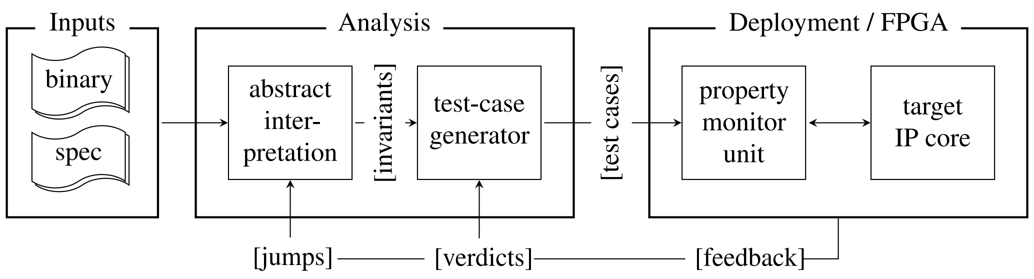

FIGURE 1. The CEVTES framework.

4 PROPERTIES OF INTEREST

Our framework supports local assertions and global invariants. The simplicity of the specification language is based on a conclusion we drew when conducting an industrial case study: temporal logics are often difficult to handle for test engineers [16]. In contrast, assertions and invariants are natural to test engineers, as they are used in everyday software development [17, p. 10].

4.1 Specification language

To express program properties of interest CEVTES supports a specification language with the following grammar:

$$\psi \quad ::= (\mathscr{A}(pc,\varphi) \mid \mathscr{I}(\varphi))^*$$

$$\begin{array}{lllll}
\varphi & : & logic & \mid & \neg logic \\
logic & : & rel & \mid & (rel \diamond rel)^* \\
rel & : & arith & \mid & (arith \star arith)^* \\
arith & : & mul & \mid & (mul \bullet mul)^* \\
mul & : & unary & \mid & (unary \circ unary)^* \\
unary & : & & & (+ \mid -)^* \; term \\
term & : & & & (r_x \mid sfr \mid constant)
\end{array}$$

The intended interpretations for the operators $\star, \bullet, \circ$ are the same as in everyday arithmetic over the domain of the integers $\mathbb{Z}$. Together with $\neg$ the $\diamond$ operator represents the standard Boolean connections. The Kleene star * is interpreted as in regular expressions and has the meaning zero or more times. In short, the supported operators are:

$$\begin{array}{l}
\diamond \in \{\wedge\ ,\ \vee\} \\
\star \in \{<\ ,\ \leq\ ,\ >\ ,\ \geq\ ,\ !=\ ,\ ==\} \\
\bullet \in \{+\ ,\ -\} \\
\circ \in \{\times\ ,\ \div\ ,\ \%\}
\end{array}$$

The property φ is an expression over memory locations $m \in \mathsf{Mem}$. The satisfaction relation associated with φ is intuitively clear, following the standard inductive definition. If $m \in \mathsf{Mem}$ satisfies φ, we write $m \models \varphi$. Properties, in turn, can be of local or global nature. A *local* assertion is a property $\mathscr{A}(pc,\varphi)$ attached to a certain program location $pc \in \mathsf{Locs}$. Given a set of states $S \subseteq \mathsf{Locs} \times \mathsf{Mem}$, then $\mathscr{A}(pc,\varphi)$ holds w.r.t. S iff $m \models \varphi$ for all $\langle pc', m\rangle \in S$ with $pc = pc'$. Similarly, a *global* invariant $\mathscr{I}(\varphi)$ holds iff $m \models \varphi$ regardless of pc'.

4.2 Restrictions on global invariants

While local assertions can be of arbitrary complexity, we restrict global invariants to be a system of logahedral constraints, which are a restricted form of two-variable-per-inequality constraints. A logahedron describes a set of points satisfying a conjunction of logahedral constraints. The logahedra abstract domain is described by Howe and King in [18] and is a feasible compromise between analysis precision and computational complexity. Constraints for the PMU are in the form of:

$$\pm 2^n \cdot r_i \pm 2^m \cdot r_j \leq C,$$

where r_i, r_j are registers $\in \mathsf{Locs}$, C a constant in $\mathbb{Z}$, and n, m are exponents in $\mathbb{Z}$. The restriction to logahedral constraints is caused mainly by hardware implementation considerations as multiplication and division demand resource hungry hardware designs in general, whereas checking of $\pm 2^n \cdot r_i \pm 2^m \cdot r_j \leq C$ can be implemented with binary adders and little combinatorial logic. Let $Add(\texttt{<a>},\texttt{<b>},\texttt{c})$ be a ripple carry adder operating on the unsigned vectors $\texttt{<a>}_U$ and $\texttt{<b>}_U$ and let c be a single bit representing the carry-in. Then, a subtracter representing $a - b$ is equivalent to $Add(\texttt{<a>},\texttt{<}\,\overline{\texttt{b}}\,\texttt{>},1)$. Similar combinations for relational operators can be found in [19, Chap. 6]. Division and multiplication by 2^n are translated to logical right and left shifts.

4.3 Frontend

As an alternative to reading a user-defined specification, our framework can also extract existing assertions from the high-level representation of the program. This is achieved by parsing compiler-generated debug information which is then further exploited to reconstruct the correspondence between memory locations and global program variables.

Listing 1 shows examples for $\mathscr{A}(pc, \varphi)$ (lines 1-7) and $\mathscr{I}(\varphi)$ (lines 9-14) properties. The local assertion *Assert1* requires that the claims *#1 - #3* hold whenever the program traverses program counter location `0x60`. Claim *#1*, e.g., expresses that the sum of registers $r7$, $r6$, $r5$ and *ACC* shall be less or equal to the value in register $r90$. The global invariant *Inv1* requires that claims *#1* and *#2* hold on all executions of the program under scrutiny.

```
 1 begin property Assert1:
 2   @PC 0x60:
 3           #1: r7 + r6 + r5 + ACC <= r90;
 4           #2: r5 + r6 <= 4;
 5           #3: (r21 == 0x80) && (P0 >= 0x90);
 6   end@;
 7 end property;
 8
 9 begin invariant Inv1:
10   @every PC:
11           #1: r7 − r6 <= 0x70;
12           #2: −2∗r5  + 4∗r6 != 0x55;
13   end@;
14 end invariant;
```

Listing 1. Example properties.

5 THE PROPERTY MONITOR UNIT

Our analysis framework derives a test suite Γ with a finite number n of test cases T. T consists of the two disjoint sets $T_{\checkmark}$ and $T_{\times}$ ($T_{\checkmark} \cap T_{\times} = \emptyset$), where $T_{\checkmark}$ are feasible test cases and $T_{\times}$ are infeasible test cases, i.e., spurious warnings.

The Property Monitor Unit (PMU) executes all the test cases in T and detects spurious test cases $T_{\times}$, thereby, successively removing infeasible test cases and reducing T to test cases residing in $T_{\checkmark}$. This relieves the system designer from performing manual valuations of all these cases, which is a tedious and error-prone task.

The PMU either reports *infeasible* (when the test case left the intended control flow given by π), *violation* (when a global invariant failed to be proved correct), or *spurious* (when the property violation found by the host analysis framework could not be affirmed by the PMU) to the host application.

Our design of the respective hardware unit was targeted to a flexible, lightweight, generic and non-intrusive solution. This led us to the following design decisions:

***Flexibility/Simplicity*:** To facilitate both, accurate and complex as well as fast and simple checks, we opted for a hybrid approach: Elementary arithmetic operations and comparison operations are performed in hardware next to the system-under-test, while more elaborate calculations are performed in software offline, based on logs of all register/RAM interactions that are collected by a monitoring unit and sent to the host. To that end, the interface must provide sufficient bandwidth so that the transfers and checks can be handled within reasonable time.

***Non-intrusiveness*:** In contrast to other approaches our PMU does not require the modification of the target hardware or software. We consider this an important advantage. Our hardware operates as an extension to the target hardware that only requires access to the data and instruction memory buses. Furthermore, provisions must be made to load the software into the targets' PROM as well as to control the executions on the target CPU (Stop/Step/Go).

***Generality*:** So far, our approach assumes (1) a strictly sequential execution of the program code, which is a very usual property for the targeted embedded controllers, and (2) access to the register file via the memory bus, for which purpose a debug interface can alternatively be used as well.

5.1 Overview

The PMU is shown in Fig. 2. A bidirectional FIFO-buffered communication with the host is managed by the USB link block, interfacing an USB 2.0 controller. Property checking is controlled by the PMU controller that coordinates the different monitoring functions. The whole design runs within an FPGA and is attached to an out-of-the-box microcontroller IP-core. Note that, we did not modify any of the behavioral code of the microcontroller. The PMU is attached to the RAM interface, the PROM interface, and the I/O interface of the IP core. The RAM interface comprehends the data and the address bus of the microcontroller. Property checking consists of an initialization and an execution phase and works as follows.

Initialization Phase:

1. The host application transfers a test case t to the PMU.
2. The PMU controller writes the binary code of the program under scrutiny into the PROM and writes the buffer of the path monitor with a list of expected program counter locations, that the test case is expected to traverse. Re-buffering is managed by the PMU controller. Similar, dedicated external inputs (e.g., port inputs or received serial bytes) and specification items are transfered to the environment control block and the invariant checker, respectively.
3. The PMU controller sets up the online property checker with the set of global invariants to be checked.
4. The initialization phase is completed when the PMU controller enables the system clock for the IP core.

Execution Phase:

- The RAM event logger is triggered by the RAM write enable signal asserted by the microcontroller. Whenever the microcontroller writes to the RAM, i.e., writes a memory location, the RAM event logger collects the actual program

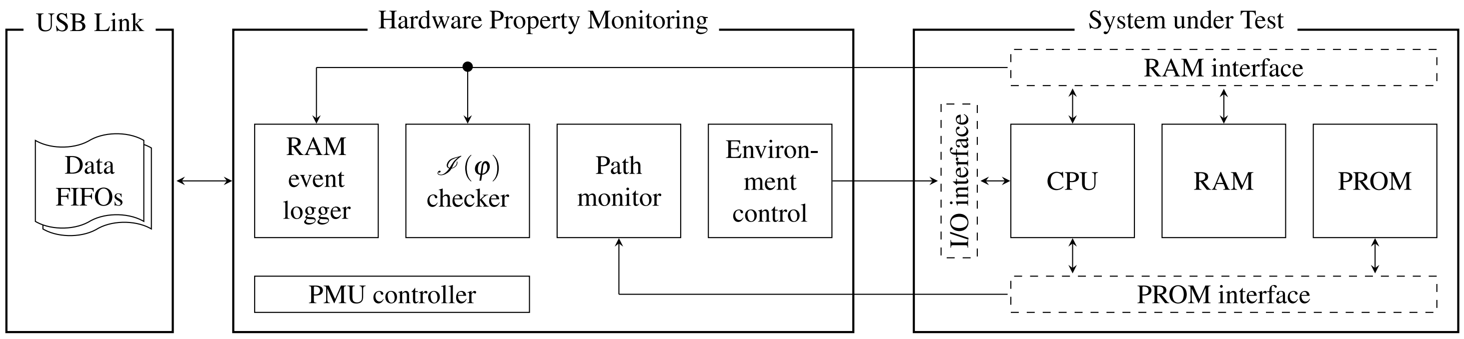

FIGURE 2. The CEVTES property monitor unit.

counter *pc*, the newly written value Δ and the destination memory location @ and forwards this information by means of a RAM update δ to the PMU controller, which takes care of the transmission to the host.
- Parallel to RAM event logging, the invariant checker checks at every progress of the program a set of global invariants. Property violations are reported back to the PMU controller.
- The path monitor takes a path of *pc* locations π as input and compares them with the actual execution of the test case by sensing the PROM interface. For various reasons, it is vital to ensure that the actual test case traverses exactly the same path as predicted by the test-case generator. First, due to the imprecision of abstract interpretation infeasible test traces may be generated. Second, in the presence of indirect jumps the path monitor helps to dynamically detect new jump targets on-the-fly, thereby, assisting CFG reconstruction as newly detected jump targets can be simply added as a new edge to the CFG, hence, subsequently refining the CFG.

5.2 Property Monitoring

The main intention of the PMU is to enable a finite set of global invariants to be monitored in a non-intrusive fashion (online monitoring). To allow more complex properties to be checked, the RAM event logger enables property checking on the host computer (offline monitoring).

Online Monitoring: Global invariants are monitored on-the-fly, thus, concurrent to the execution of a test case at the target microcontroller. The superior advantage of this approach is that the microcontroller may run at its full system clock speed and does not need to be sent to a "halt" state when a property is checked. This is very useful for frequent but simple checks.

Offline Monitoring: More elaborate properties and local assertions are checked at the host computer. The PMU collects successive RAM updates δ and offers them to the host. δ is a triple $\langle\Delta, @, pc\rangle$, where Δ is the data update, @ the address of the memory location, and *pc* a program counter location. Given a strict sequential execution of program code, RAM updates are in a temporal order. For example, $\delta := \langle 0x8, 0x15, 0xC1C1\rangle$ is read as: memory location $0x15$ becomes $0x8$ at program location $0xC1C1$.

The host application starts with an initial program state $S_0\langle pc, m\rangle$ and successively applies the update $\psi(\delta)$ to derive the next state $S'\langle pc', m'\rangle$. For sake of simplicity we rewrite $\delta\langle\Delta, @, pc\rangle$ to $\langle\delta_\Delta, \delta_@, \delta_{pc}\rangle$ and $S\langle pc, m\rangle$ to $\langle S_{pc}, S_m\rangle$. $S_{m[i]}$ refers to the memory location m with address i (ranges over all bytes in the RAM of the target microcontroller, i.e., $0 \leq i < |\mathsf{Locs}|$). Then, an update $S \xrightarrow{\psi(\delta)} S'$ is defined as follows:

$$S \xrightarrow{\psi(\delta)} S' = \begin{cases} S'_{pc} & := \delta_{pc} \\ S'_{m[i]} & := \begin{cases} \delta_\Delta & \text{if } i \equiv \delta_@, \\ S_{m[i]} & \text{otherwise.} \end{cases} \end{cases}$$

5.3 Runtime Feedback

CFG reconstruction: A prerequisite for sound abstract interpretation requires that a CFG is present. However, under the presence of indirect control this is a notorious hard problem [20]. Consequently, our abstract interpretation framework starts with an initial, incomplete CFG that is incrementally refined as test-case execution advances. Whenever a new jump target is detected, we add a new edge in the CFG, thus, subsequently refining the CFG. As mentioned before, the path monitor compares the list π of expected program counter locations – which are equivalent to vertices in the CFG – to the program counter locations traversed by the test case. Whenever the path monitor encounters a deviation and the last executed instruction was an indirect jump (`iJMP`), an edge from vertex v_{n-1} to vertex v_n is inserted, where v_{n-1} is the location of `iJMP` and v_n is the newly detected jump target.

Verdicts: Whenever one of the global invariant checkers $\mathscr{I}(\varphi)$ witnesses a property violation, test-case execution and IP-core are first kept in a halt state. To facilitate further analysis, the violation is reported back to the host together with the current *pc* location and the valuation of the registers involved in property φ before the next test case t in Γ is executed.

6 WORKED EXAMPLE

In this section we show how the introduced PMU supports testing and monitoring of specification items in industrial embedded code. The embedded software of a cooling controller for a power converter serves as a running example.

6.1 System Overview

The target embedded system is a cooling controller that monitors current and voltage parameters of a DC/DC flyback converter [21] in an industrial setting and adapts the system's cooling according to the power dissipation. A system overview is given in Fig. 3.

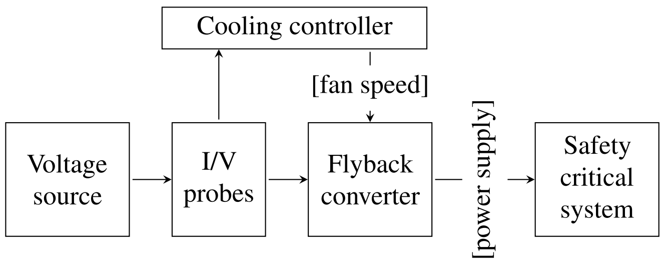

FIGURE 3. The Cooling Controller Application.

6.2 The Flyback Converter Specification

The design of the flyback converter puts restrictions on input voltage, input current, and power dissipation. For example, the switch at the primary side of the converter owns a certain nominal switching-voltage and the coil has a nominal operating current. These requirements can be summarized as a system of inequalities:

$$\begin{array}{llll} \text{Req1:} & 0A & \leq I & \leq 5A \\ \text{Req2:} & 0V & \leq V & \leq 5V \\ \text{Req3:} & 0W & \leq V \times I & \leq 16W \end{array}$$

Note that Req3 is equivalent to $0W \leq P \leq 16W$, and expresses the valid power dissipation range. In geometry, this inequalities are described by four closed half-planes, relating to the minima and maxima values for current and voltage, and a rational function ($f(x) = \frac{P_{max}}{x}$) describing the power dissipation limitation. The flyback converter operates within the specification as long as the current and the voltage remain within the hatched area of Fig. 4.

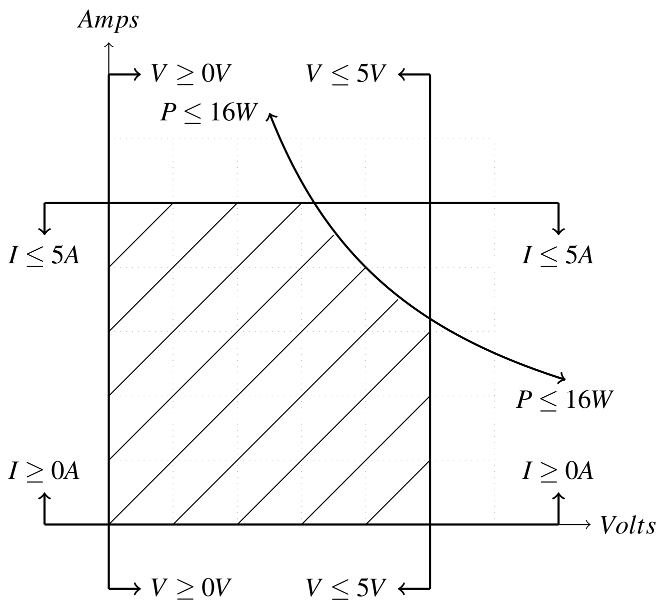

FIGURE 4. FlyBack Converter Specification (exact).

6.3 Target Application

The target application's purpose is to adjust the fan speed of the DC/DC converter according to the measured power consumption. With every iteration of the main loop, the software reads values from two ADCs, i.e., the I/V probes in Fig. 3, one corresponding to the voltage measured at the input of the flyback converter, the other measures a voltage drop across a shunt resistance to measure the input current. There are different scaled-down versions of the product, thus, shunt resistors of different resistance are shipped. This necessitates that the software receives the actual shunt resistance value during startup from a supervising industrial control system.

Source Code: Lets consider the code snippet in Lst. 2. The *main* method calls *sysConfigure()*, where the target application requests the shunt resistance value from the overlaying control system. *processTask()* summarizes the tasks assigned to the target application. For brevity, the code is not presented in length here as it does not contribute in demonstrating our approach. Within the while loop the function *adjustCooling()* is called, that reads two ADC samples (lines 10 and 11), one for voltage and one for the voltage drop across a shunt resistor. Next, the actual current is calculated with application of Ohm's law (line 12). Finally, based on the calculated power dissipation, the cooling system is set to one of the states in $\{$*off, moderate, full*$\}$.

```
 1 UINT8 sysVoltage;
 2 UINT8 sysCurrent;
 3 UINT8 sysPowerDissipation;
 4 UINT8 shuntResistance;
 5
 6 UINT8 adjustCooling(UINT8 shuntResistance){
 7   UINT8 shuntVoltage;
 8   UINT8 returnVal = C_SUCCESS;
 9
10   sysVoltage = readADCPort(PORT_V_ADC);
11   shuntVoltage = readADCPort(PORT_I_ADC);
12   sysCurrent = shuntVoltage/shuntResistance;
13   sysPowerDissipation = sysCurrent * sysVoltage;
14
15   if(sysPowerDissipation <= POWER_LOW){
16       returnVal = setCoolingMode(OFF);
17   }
18   else if(sysPowerDissipation <= POWER_MID){
19       returnVal = setCoolingMode(MODERATE);
20   }
21   else
22       returnVal = setCoolingMode(FULL);
23   }
24
25   return returnVal;
26 }
27
28 void main(void){
29   sysConfigure(&shuntResistance);
30   while(1) {
31       if(adjustCooling(shuntResistance) == E_FATAL){
32           sysReset();
33       }
34       processTask();
35   }
36 }
```

Listing 2. Source code under scrutiny.

Debug Information: From the compiler (we use Keil μVision 3 v3.23) generated debug information, we can (automatically) derive the following correspondence between memory addresses and global variables in the C-code.

C-Code Variable	Type	Memory Addr.	Symbol
sysPowerDissipation	UINT8	0000H	r_0
sysVoltage	UINT8	0001H	r_1
sysCurrent	UINT8	0002H	r_2
shuntResistance	UINT8	0003H	r_3

6.4 Abstract Interpretation Findings

We run our abstract interpretation on the binary code of Listing 2. The analysis is able to disprove invariant Req3 and reveals a major problem in the code.

Possible divide by zero: The method *adjustCooling()* obtains the value for the *shuntResistance* from the call to *sysConfigure()* in the *main* loop. Note that, *sysConfigure()* requests the value of the shunt resistor from the overlaying industrial control system. Suppose there is a misconfiguration in the control system and the value for the *shuntResistance* erroneously takes on the value `0x00`, then, the calculation in line 12 causes a divide by zero.

At the Intel MCS-51 architecture a division by zero is a particular tricky issue. For the `DIV A ÷ B` instruction, the datasheet [22] states:

> *If B had originally contained* `0x00`*, the values returned in the Accumulator and B-register will be undefined and the overflow flag will be set. The carry flag is cleared in any case.*

However, the exact interpretation of this corner case remains with the IP-core designer. Considering the actual register level design of the divide instruction, the three most likely implementation options are:

(1) The Accumulator is reset to `0x00`, thus, regardless of the actual inputs, the cooling system will always be turned off in line 16. Thus the error is likely to go unnoticed in normal operation as long as additional cooling is not required.
(2) The Accumulator remains unchanged (i.e., keeps the last value), thus, the variable *sysCurrent* will be assigned the value of the *shuntVoltage* calculated on line 12.
(3) The Accumulator is assigned a random value, i.e., the variable *sysCurrent* is random too, causing unpredictable cooling behavior.

For the presented case study, we attached our monitoring unit to the freely available Intel MCS-51 IP-core of Oregano Systems [23], which implements (2).

Error Scenario: Suppose the system actually uses a shunt resistance of 2 Ohms and the overlying control system wrongly sends a shunt resistance value of `0x00` to the target application. Further, suppose that the current consumed by the flyback converter is 3 Amps, thus, well within its safe operational range. Then the measured voltage across the shunt resistor equals $6V$. The described division by zero occurs and wrongly assigns the value of 6 to the variable *sysCurrent* in code line 12.

Consequently, with a measured *sysVoltage* of $4V$, the determined power dissipation in line 13 evaluates to $6A \times 4V = 24W$, which clearly exceeds the maximum power dissipation of $16W$.

Derived test case: Finally, the test-case generator will derive a test case with the following semantics:

```
@ 6    adjustCooling(shuntResistance ↤ 0)
@ 10           4 ↤ readADCPort(PORT_V_ADC)
@ 11           3 ↤ readADCPort(PORT_I_ADC)
@ 13   assert(sysPowerDissipation <= 16)
```

In the remainder of this section we will demonstrate how this test case can be automatically classified as real bug by use of the PMU on the target platform.

6.5 Test-case validation

As reported in Section 4, the CEVTES framework supports online and offline monitoring of specification items, which can be conveniently applied here.

Offline Monitoring Offline monitoring allows arbitrary arithmetic and logic connections among atomics, thus, we can state the specification items in a 1:1 manner (List. 3).

```
begin offline invariant FlyBackConverter:
  @every PC:
          #1: r2 <= 5;
          #2: r2 >= 0;
          #3: r1 <= 5;
          #4: r1 >= 0;
          #5: r0 <= 16;
  end@;
end offline invariant;
```

Listing 3. Invariants for offline monitoring.

Invariants #1...#4 relate to the requirements Req1 and Req2, whereas #5 covers the power dissipation limitation, i.e., Req3. These invariants will be checked with every memory update δ sent by the RAM event logger of the PMU (cf. Fig 2), and need a host computer to be evaluated. By applying the derived test-case, the property violation proves true, thus, the found warning is a true error.

Online Monitoring The flyback converter specification can also be tracked online, i.e., during the operation of the embedded system, without the intervention of a host computer. This is done by the invariant checker of the PMU and can handle the exact same invariants as shown for the offline monitor.

Nevertheless, the properties postulated so far, are delusive particularly with regard to Req3 as they make no use of the relational characteristics among current and voltage to express the power dissipation limits. Instead of claiming that $r_0 \leq 16$, we could claim that $r1 \times r2 \leq 16$, thus, the evaluation does not rely on the power dissipation calculated by the microcontroller application.

Whereas requirements Req1 and Req2 can be directly expressed, the rational function of the power dissipation limitation is approximated by a linear inequality to meet a logahedral property for the invariant checker, (cf. Sect. 4). This inequality is a half plane defined by a straight line $I = k \times V + d$ through the points $P_A(3,5)$ and $P_B(5,3)$, with a slope of $k = \frac{3-5}{5-3} = -1$ and an y-intercept of 8, leading $I + 1 \times V \leq 8$. The preprocessed requirements 1...3 for the hardware invariant checker unit are stated in Lst. 4. Note that the invariants now take into account the arithmetical connection between r_1 and r_2, i.e., current and voltage.

```
begin online invariant FlyBackConverter:
  @every PC:
          #1: r2 <= 5;
          #2: r2 >= 0;
          #3: r1 <= 5;
          #4: r1 >= 0;
          #5: r2 + 1*r1 <= 8;
  end@;
end online invariant;
```

Listing 4. Invariants for online monitoring.

Geometrically speaking, the final specification consists of five half planes confining a convex polygon in two dimensions, as shown in Fig 5.

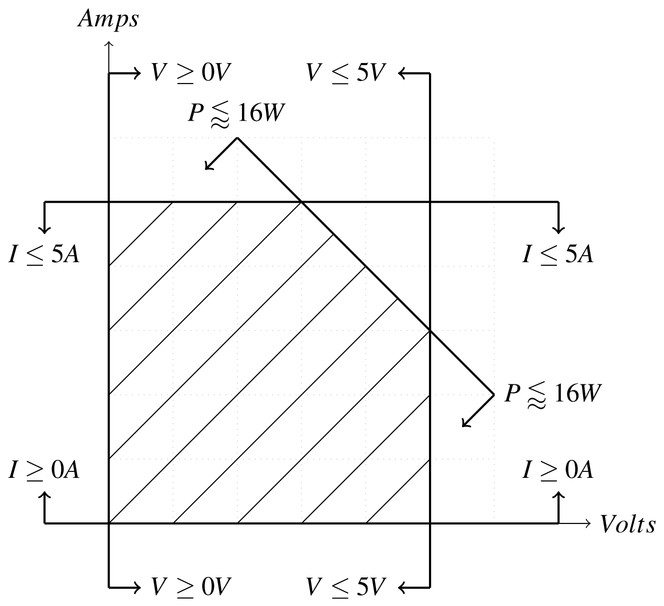

FIGURE 5. FlyBack Converter Specification (approximation).

When executing the test-case the PMU correctly detects the property violation in a non-intrusive and passive way, thus, property checking is parallel to the execution of the test-case, where the IP-core runs at its nominal clock speed, i.e. 8 MHz. The property violation is reported to the host computer, where the test-case generation took place. However, the PMU may also work in a self-sufficient mode and stay attached to the IP-core – even after the testing and verification phase of the product – and serve as concurrent oracle.

7 CONCLUSION

In this paper, we have argued that combining formal verification approaches with testing may pave the way for exhaustive tests of embedded system software. In practice, domain knowledge is required to extract an abstract model from a system in order to keep the number of states manageable. A test engineer may then verify given properties or be able to disprove them. Property violations, however, may turn out as "false negatives" since they were derived from an abstract model rather than the concrete system. The contribution of this paper is a method that either automates the verification of these violations or discards them. We introduced a property monitor unit that supports checking properties on the fly, i.e., during normal operation or test-case execution of the target system. The property monitor can easily be attached to an industrial IP-core running on an FPGA.

As future work, we will try to improve the flexibility of the on-line checks in hardware, allowing for more complex invariants to be checked dynamically. Furthermore, we plan to conduct some real-world case studies in order to assess the potential of this solution, possibly based on our earlier experiences with verifying industrial embedded software [16].

ACKNOWLEDGMENT

The work of Thomas Reinbacher and Andreas Steininger has been supported within the FIT-IT project CEVTES managed by the Austrian Research Agency FFG under grant 825891. The work of Jörg Brauer and Stefan Kowalewski has been, in part, supported by the DFG Cluster of Excellence on *Ultra-high Speed Information and Communication*, German Research Foundation grant DFG EXC 89, and the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*.

REFERENCES

[1] Baier, C., and Katoen, J.-P., 2008. *Principles of Model Checking*. The MIT Press. ISBN 026202649X.

[2] Cousot, P., and Cousot, R., 1977. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In POPL, pp. 238–252.

[3] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S., 2010. "seL4: formal verification of an operating-system kernel". *Commun. ACM, **53***(6), pp. 107–115.

[4] Clarke, E. M., and Veith, H., 2004. "Counterexamples revisited: Principles, algorithms, applications". In Verification: Theory and Practice, Vol. 2772 of *LNCS*, Springer, pp. 41–43.

[5] Rival, X., 2005. "Understanding the origin of alarms in Astrée". In *SAS*, Vol. 3672 of *LNCS*. Springer, pp. 303–319.

[6] Heimdahl, M. P. E., George, D., and Weber, R., 2004. "Specification test coverage adequacy criteria = specification test generation inadequacy criteria?". In HASE, IEEE, pp. 178–186.

[7] Eide, E., and Regehr, J., 2008. "Volatiles are miscompiled, and what to do about it". In EMSOFT, ACM, pp. 255–264.

[8] Havelund, K., 2008. "Runtime verification of C programs". In TestCom/FATES, Springer, pp. 7–22.

[9] Colin, S., and Mariani, L., 2005. *Model-Based Testing of Reactive Systems*. Springer, ch. Run-Time Verification, pp. 525–555.

[10] Tsai, J. J. P., Fang, K.-Y., Chen, H.-Y., and Bi, Y.-D., 1990. "A noninterference monitoring and replay mechanism for real-time software testing and debugging". *IEEE Trans. Softw. Eng., **16***, pp. 897–916.

[11] Drechsler, R., 2003. "Synthesizing checkers for on-line verification of system-on-chip designs". In ISCAS, Vol. 4, pp. 748–751.

[12] Mercer, E., and Jones, M., 2005. "Model checking machine code with the GNU debugger". In SPIN Workshop on Model Checking Software, Vol. 3639 of *LNCS*, pp. 251–265.

[13] Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., and Kowalewski, S., 2011. "Test-case generation for embedded binary code using abstract interpretation". In MEMICS'10 – Selected Papers, Vol. 16 of *OASIcs*, Schloss Dagstuhl, pp. 101–108.

[14] Brauer, J., and King, A., 2010. "Automatic abstraction for intervals using boolean formulae". In SAS, Vol. 6337 of *LNCS*, Springer, pp. 167–183.

[15] Cousot, P., and Cousot, R., 1976. "Static determination of dynamic properties of programs". In 2nd International Symposium on Programming, pp. 106–130.

[16] Reinbacher, T., Horauer, M., Schlich, B., Brauer, J., and Scheuer, F., 2011. "Model checking embedded software of an industrial knitting machine". *IJITCC, **1***(2), pp. 182–205.

[17] Hoare, C., 2003. "Assertions: A personal perspective". *IEEE Annals of the History of Computing, **25***, pp. 14–25.

[18] Howe, J., and King, A., 2009. "Logahedra: A new weakly relational domain". In *ATVA*, Vol. 5799 of *LNCS*. Springer Berlin / Heidelberg, pp. 306–320.

[19] Kroening, D., and Strichman, O., 2008. *Decision Procedures: An Algorithmic Point of View*. Springer. ISBN 3540741046.

[20] Kinder, J., Veith, H., and Zuleger, F., 2009. "An abstract interpretation-based framework for control flow reconstruction from binaries". In VMCAI, Vol. 5403 of *LNCS*, Springer, pp. 214–228.

[21] Billings, K., 1999. *Switchmode Power Supply Handbook*. McGraw-Hill Professional. ISBN 0070067198.

[22] Intel Cooperation, 1994. *MCS 51 Microcontroller Family User's Manual*. Order No.: 272383-002.

[23] Oregano Systems, 2002. MC8051 IP core user guide. http://oregano.at/services/8051.htm.