Past Time LTL Runtime Verification for Microcontroller Binary Code

Thomas Reinbacher[1], Jörg Brauer[2], Martin Horauer[3], Andreas Steininger[1], and Stefan Kowalewski[2]

[1] Embedded Computing Systems Group, Vienna University of Technology, Austria
[2] Embedded Software Laboratory, RWTH Aachen University, Germany
[3] Department of Embedded Systems, UAS Technikum Wien, Austria

Abstract. This paper presents a method for runtime verification of microcontroller binary code based on past time linear temporal logic (ptLTL). We show how to implement a framework that, owing to a dedicated hardware unit, does not require code instrumentation, thus, allowing the program under scrutiny to remain unchanged. Furthermore, we demonstrate techniques for synthesizing the hardware and software units required to monitor the validity of ptLTL specifications.

1 Introduction

Program verification deals with the problem of proving that all possible executions of a program adhere to its specification. Considering the complexity of contemporary embedded software, this is a particularly challenging task. Conventional ad-hoc testing is significantly less ambitious; it is thus the predominant method in the (embedded) software industry. Typically, a set of test-cases is derived manually or automatically in a best effort fashion. Then, the arduous task of judging the results of a test-case run often remains with the test engineer.

1.1 Runtime Verification by Code Instrumentation

The field of runtime verification [7] has gained momentum as it links traditional formal verification and monitoring the execution of test cases. The aim is to increase confidence in correctness of the system, without claiming freedom from defects. In runtime verification, test oracles, which reflect the specification, are either automatically derived (e.g., from a given temporal logic formula that specifies a requirement) or formulated manually in some form of executable code. Correctness of an execution is then judged by means of evaluating sequences of events observed in an instrumented version of the program under scrutiny. Instrumentation can either be done manually, or automatically by scanning available program nodes (e.g., assignments, function calls, ...) at the level of the implementation language. Function calls are then inserted to emit relevant events to an observer, i.e., the test oracle. The latter approach has proven feasible for high-level implementation languages such as C, C++, and Java, as well as for hardware description languages such as VHDL and Verilog. Various runtime verification frameworks have thus emerged [6, 8, 13, 14, 20].

1.2 Pitfalls of Code Instrumentation

Despite considerable technical progress, existing approaches to runtime verification are not directly transferable to the domain of embedded systems software, mainly due to the following reasons:

1. Embedded code often adopts target-specific language extensions, direct hardware register and peripheral access, and embedded assembly code. When instrumenting such a code basis, one has to take all the particularities of the target system into account, depleting the prospect of a universal approach.
2. In its present shape runtime verification proves the correctness of high-level code. However, to show that a high-level specification is correctly reproduced by the executable program, it is necessary to verify the translation applied to the high-level code as it is not unknown for compilation to introduce errors [2, 9, 24]. One thus needs to prove that for a given source code P, if the compiler generates a binary code B without compilation errors, then B behaves like P [29]. Proving correctness of the compiler itself is typically not feasible due its complexity and its sheer size [21, 22]. Flaws introduced by the compiler may thus remain unrevealed by existing approaches.
3. Instrumentation at binary code level is never complete as long as the full control flow graph (CFG) is not reconstructed from the binary program. Although CFG reconstruction of machine code is an active research area [3, 12, 17], generating sound yet precise results remains a challenge.
4. Instrumentation increases memory consumption, which may be of economical relevance for small-sized embedded targets.

We conclude that a non-instrumenting approach for microcontroller binary code may be a notable contribution to further establish the use of lightweight formal techniques, such as runtime verification, in the embedded software industry.

1.3 Requirements to Runtime Verification of Microcontroller Code

To overcome the pitfalls discussed so far, which prohibit the application of existing frameworks to the embedded systems domain, it is necessary to provide a framework that works on the level of binary code and additionally satisfies the following requirements:

Req1: Generality For a verification on the binary code level the target microcontroller must be fixed. However, the approach shall not be bound to a certain compiler (version) or high-level programming language.

Req2: No Code Instrumentation Typically, software event triggers are instrumented to report execution traces as sequences of observations. For small-scale embedded platforms, it is necessary to extract event sequences without code instrumentation.

Req3: Provide Mechanics to Evaluate Atomic Propositions The atomic propositions (AP) of the specification need to be evaluated on microcontroller states of the running system. We need to find a reasonable trade-off between expressiveness of the AP and the complexity of their evaluation.

Furthermore, to be useful in an industrial environment, some practical requirements need to be considered:

Req4: Automated Observer Synthesis From a user point of view, it is desirable to input a specification in some (temporal) logic, which is automatically synthesized into an observer that represents the semantics of the temporal property.

Req5: Usability We aim at a framework which is applicable in industrial software development processes; at best, this is a push-button solution. It shall be possible to include implementation-level variables in the specification, which are automatically mapped to the memory state on the target hardware.

1.4 Contributions to Runtime Verification

The contribution of this paper is a framework for supervising past time linear temporal logic (ptLTL) properties [15] in embedded binary code. ptLTL allows to specify typical requirements to embedded software in a straightforward fashion, which contrasts with our experiences using Computation Tree Logic (CTL) [31]. Further, we present a host application that interacts with a customized hardware monitoring unit and a microcontroller IP-core (executing the software under scrutiny), both of which are instantiated within an FPGA. In our approach, supervision of ptLTL specifications can take place either *offline* (using the host application) or *online* in parallel to program execution. Both options come along without any kind of code instrumentation or user-interaction. We implemented the presented approach into our testing framework called CevTes [30].

1.5 Structure of the Paper

The presentation of our contributions is structured as follows. In Sect. 2, we present preliminaries used throughout the paper. Sect. 3 introduces our framework for runtime verification of binary code. We apply our approach to a real-life example in Sect. 4. We put our work in context with related work in Sect. 5 and conclude with a discussion of achievements in Sect. 6.

2 Preliminaries

This section introduces notations used in the remainder of the paper, including a formal microcontroller model and the finite-trace temporal logic ptLTL.

2.1 Formal Microcontroller Model

Addressing memory locations: Let $\mathsf{Addr} = \{0 \leq x < |\mathsf{Mem}| \ : \ x \in \mathbb{N} \cup \{0\}\}$ denote the set of memory locations of the microcontroller, where Mem represents the (linear) address space of the microcontroller memory. We write r_x to address a specific memory location, e.g. , r_{20} denotes the memory location with address 20. We assume a memory mapped I/O architecture (e.g. Intel MCS-51), thus, I/O registers reside within Mem.

State of the microcontroller program: In the following, let $\mathbb{N}_k = \{0, \ldots, k-1\}$. A state S of the microcontroller is a tuple $\langle pc, m\rangle \in \mathsf{Locs} \times (\mathsf{Addr} \to \mathbb{N}_{2^w})$, where Locs is a finite set of program counter values, and $m : \mathsf{Addr} \to \mathbb{N}_{2^w}$ is a map from memory locations (with bit-width w) to memory configurations. The state space of the program is thus a subset of $\mathsf{Locs} \times (\mathsf{Addr} \to \mathbb{N}_{2^w})$. We denote the initial microcontroller state S_0 by $\langle \texttt{0x00}, m_0\rangle$ where m_0 represents the configuration of all memory locations after power-up and `0x00` is the assumed reset vector.

State updates: State updates trigger a state transition, thereby, transforming a predecessor state S^{-1} into the current state S. A state update is a triple $\delta = \langle \zeta_\delta, @_\delta, pc_\delta\rangle$, where ζ_δ is the new configuration of the altered memory location, $@_\delta$ is its address, and pc_δ is the new program counter value. Given a strict sequential execution of the program, state updates are in temporal order. A state update $S^{-1} \xrightarrow{\delta} S$ transforms $S^{-1} = \langle pc^{-1}, m^{-1}\rangle$ into $S = \langle pc_\delta, m\rangle$ where:

$$m(i) = \begin{cases} \zeta_\delta & \text{if } i = @_\delta \\ m^{-1}(i) & \text{otherwise} \end{cases}$$

A sequence of events, denoted π, is a trace of state updates δ, e.g. , $\pi = \langle \delta_0 \ldots \delta_n\rangle$.

2.2 Past Time LTL

While past time operators do no yield extended expressive power of future time LTL [10, Sect. 2.6], a specification including past time operators may sometimes be more natural to a test engineer [19, 23]. A ptLTL formula ψ is defined as

$$\begin{aligned} \psi ::= & \mathit{true} \mid \mathit{false} \mid \mathit{AP} \mid \neg\psi \mid \psi \bullet \psi \\ & \odot\psi \mid \diamond\psi \mid \boxdot\psi \mid \psi\ S_s\ \psi \mid \psi\ S_w\ \psi \end{aligned}$$

where $\bullet \in \{\wedge, \vee, \rightarrow\}$. $\odot\psi$ means *previously* ψ, i.e., it is the past-time analogue of next. Likewise, the other temporal operators are defined as: $\diamond\psi$ expresses *eventually in the past* ψ and $\boxdot\psi$ is referred to as *always in the past*. The duals of the until operator are S_s and S_w, i.e. , *strong since* and *weak since*, respectively.

Monitoring operators: These basic operators can be augmented by a set of monitoring operators [15, 20]. The semantics of the monitoring operators is derived from the set of basic operators in ptLTL, thus, do not add any expressive power. However, they provide the test engineer a succinct representation of the most common properties emerging in practical approaches:

$$\psi ::= \uparrow \psi \mid \downarrow \psi \mid [\psi, \psi)_s \mid [\psi, \psi)_w$$

$\uparrow \psi$ stands for *start* ψ (i.e., ψ was false in the previous state and is true in the current state, equivalent to $\psi \wedge \neg \odot \psi$), $\downarrow \psi$ for *end* ψ (ψ was true in the

previous state and is false in the current state, equivalent to $\neg\psi \wedge \odot\psi$), and $[\psi_1, \psi_2)$ for *interval* ψ_1 ψ_2 (ψ_2 was never true since the last time ψ_1 was true, including the state when ψ_1 was true, equivalent to $\neg\psi_2 \wedge ((\odot\neg\psi_2)\ S\ \psi_1)$). The set of atomic propositions AP contains statements over memory locations in Locs. Space constraints force us to refer the reader to [10,15,20] for a formal semantics.

Determining satisfaction: It is important to appreciate that satisfaction of a ptLTL formula can be determined along the execution trace by evaluating only the current state S and the results from the predecessor state S^{-1} [15].

3 System Overview

The following section details our runtime verification framework, as depicted in Fig. 1, which works on microcontroller binary code rather than a high-level representation of the program, thus meeting Req1.

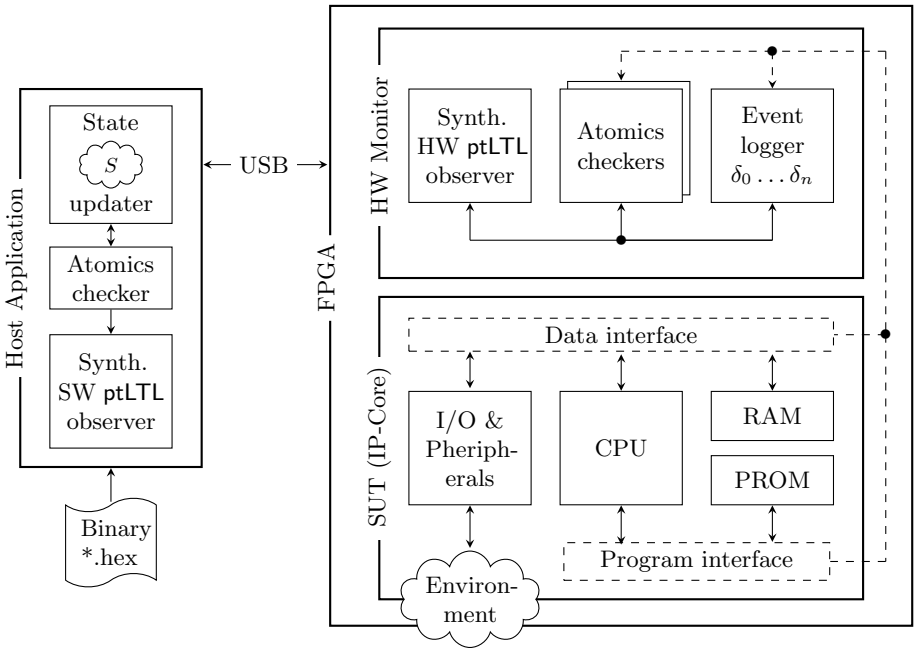

Fig. 1. System overview

We address Req2 by a hardware monitor unit, which is transparently attached to an industrial microcontroller IP-core running on an FPGA. The monitor allows

to extract execution traces without code instrumentation. We tackle Req3 by a twofold approach: (i) Offline mode: We permanently send state updates δ from an hardware implemented event logger to a host application which applies δ to the current state S^{-1} to obtain the successor state S. The AP of the ptLTL formula ψ are evaluated on S and the validity of ψ is decided by a synthesized SW ptLTL observer. (ii) Online mode: As a self-contained alternative, we check the AP of the ptLTL formula on-the-fly and decide the validity of ψ by a synthesized HW ptLTL observer directly on the FPGA.

We meet Req4 by instantiating an algorithm described by Havelund and Roşu [15], i.e., we generate observer for ptLTL as executable Java or VHDL code. Finally, we comply with Req5 by providing a graphical interface to the system and an optional debug file parser allowing to state formulas over high level symbols.

3.1 ptLTL Observer Synthesis

Runtime verification requires an observer to be attached to the system under test. Our approach supports a full FPGA based solution as well as a combined one where a hardware event logger stimulates a Java class on the host computer. Technically speaking, we derive (a) Java classes and (b) VHDL entities, both representing an observer for the specification ψ. In a subsequent step, (a) is compiled into executable Java code and (b) is synthesized into a netlist. Both observers rely on the hardware monitor unit to evaluate the AP of ψ. Whereas (a) utilizes event updates about the state of the microcontroller, (b) makes use of a dedicated atomics checker hardware unit.

Observer synthesis thus consists of the following stages: (i) We use the ANTLR parser generator [26] to parse a ptLTL formula ψ, which yields an abstract syntax tree (AST) representing the specification. (ii) After some preprocessing of the AST, we determine the n subformulas $\psi_0 \dots \psi_n$ of ψ using a post-order traversal of the AST. (iii) We generate observers as executable Java or synthesizable VHDL code [15].

3.2 Hardware Monitor Unit

The hardware monitor unit (cf. Fig. 1) is attached to the system under test, an (unmodified) off-the-shelf microcontroller IP-core[1], which is embedded into its application environment. The observer consists of three main components, namely an event logger, an atomics checker unit, and a synthesized ptLTL observer. The remainder of this section discusses the details of these components.

Event logger The event logger wiretaps the data and the program interface of the microcontroller and collects memory updates δ non-intrusively. For example, if the currently fetched instruction is `MOV [*20, 0x44]`, which moves the constant value `0x44` into r_{20}, and the current program counter pc equals `0xC1C1`, then the event logger assembles a new state update $\delta = \langle \texttt{0x44}, 20, \texttt{0xC1C1} \rangle$.

[1] For our actual implementation we employ an Intel MCS-51 IP-core from Oregano Systems (http://www.oregano.at).

Atomics checkers The purpose of these units is to check the atomic propositions of ψ, one per unit. Ideally, we would favor a full-fledged hardware-only solution allowing for arbitrary atomic propositions to be checked on-the-fly. However, as we aim at a lightweight monitor with small area overhead, we opted for offering two implementation variants: a software-implemented offline checker supports arbitrary expressions for atomics, and we use constraints similar to Logahedra [16] for the hardware-based online approach, thus allowing to establish a balance between hardware complexity and expressiveness. More specifically, the hardware-based atomics checker supports conjunction of restricted two-variable-per-inequality constraints of the form

$$(\pm 2^n \cdot r_i \pm 2^m \cdot r_j) \bowtie C$$

where $r_i, r_j \in \mathsf{Mem}$, $C \in \mathbb{Z}$, $n, m \in \mathbb{Z}$, and $\bowtie \in \{=, \neq, \leq, \geq, <, >\}$. The second operand is optional, thus allowing range constraints of the form $\pm(2^n) \cdot r_i \bowtie C$.

Fig. 2 shows the generic hardware design to evaluate a single atomic proposition. The unit is connected to the data interface. We instantiate one such unit for each $ap \in AP$; the derived verdicts $atomic(0 \ldots |AP|)$ serve as input for the ptLTL observer. The *constant* C is loaded into the compare unit; *mode* constitutes control signals to determine the operation to be performed on the operands. The write-enable signal issued by the CPU triggers the atomics checker unit which stores the value on the data bus in a register iff the destination address equals i or j, respectively. The shifter unit supports multiplication and division by 2^n. The arithmetic unit is a full-adder, serving both as adder and subtracter. Observe that, when $\mathsf{Add}(\langle \mathsf{a} \rangle, \langle \mathsf{b} \rangle, \mathsf{c})$ is a ripple carry adder for arbitrary length unsigned vectors $\langle \mathsf{a} \rangle$ and $\langle \mathsf{b} \rangle$ and c the carry in, then a subtraction of $\langle \mathsf{a} \rangle - \langle \mathsf{b} \rangle$ is equivalent to $\mathsf{Add}(\langle \mathsf{a} \rangle, \langle \overline{\mathsf{b}} \rangle, 1)$. Relational operators can be built around adders in a similar way [18, Chap. 6].

Synthesized HW ptLTL observer The synthesized ptLTL observer unit subsumes the verdicts of the diverse atomic checker units over the respective AP of ψ into a final decision $\pi \models \psi$. While in the offline mode this function is performed in software, a dedicated hardware block is needed for the online mode.

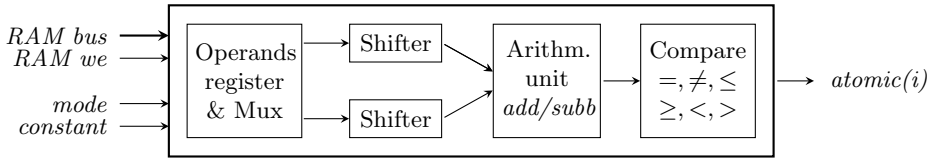

Fig. 2. The atomics checker unit

Housekeeping The hardware monitor unit supports writing the `*.hex` file under scrutiny into the target system's PROM and handles communication tasks between FPGA and host application using a high-speed USB 2.0 controller.

3.3 Host Application

The host application is responsible for offline runtime verification. It reads a `*.hex` binary file and a ptLTL formula ψ. Optionally, compiler-generated debug information is parsed and symbols in the high-level implementation language are related to memory addresses in microcontroller memory. Rather than expressing properties over memory locations within the RAM of the microcontroller, this approach allows high-level implementation symbols to be included in the formula. For example, the formula $\uparrow \texttt{foo} = 20$ is satisfied iff the memory location that corresponds to the variable `foo`, say, r_{42}, does not equal 20 in the predecessor state S^{-1} and equals 20 in the current state S. Therefore, even though the analysis is based on binary code, it is possible to state propositions over high-level symbols, which eases the process of specifying desired properties.

State updates State transitions $S^{-1} \xrightarrow{\delta} S$ are performed on each state update δ, received from the event logger. Incoming events are categorized as follows: (i) events that perform plain state updates and (ii) events that alter memory locations used in atomic propositions of the formula. Events in (i) are used to keep a consistent representation of the current microcontroller state, whereas events in (ii) additionally trigger the SW ptLTL observer to derive a new verdict.

Event evaluation Atomic propositions are directly evaluated on the current state S, and the resulting verdicts are then forwarded to the observer that decides the validity of formula ψ.

Synthesized SW ptLTL observer Whenever the destination address @ of a state update δ matches any memory location in the atomic propositions AP of ψ, the generated software ptLTL observer code is executed and a new verdict is derived. If the property is violated, the unit reports "$\times$" to the user. State updates are in temporal order, thus, it would be possible to store a sequence $\langle \delta_1, \ldots, \delta_n \rangle$ of state updates and apply the observer afterwards, decoupled from program execution. However, in our experiments, it was always possible to evaluate the events without time-penalty, i.e., as they occur while the program is running. The stored state space consists only of the current state S.

4 Worked Example

In the remainder of this section, we report on applying our toolset to embedded C code. As an example, we consider a function block specified by the PLCopen

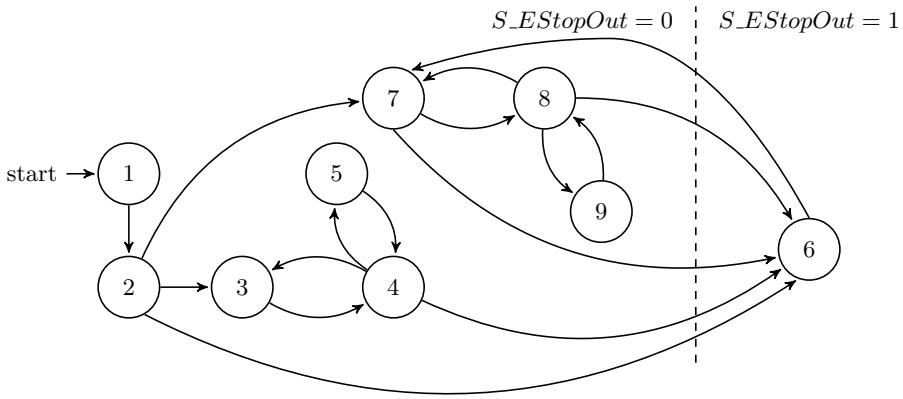

Fig. 3. The *emergency stop* function block as nondeterministic finite state machine

consortium, which has defined safety-related aspects within the IEC 61131-3 development environment to support developers and suppliers of Programmable Logic Controllers (PLC) to integrate safety-related functionality into their systems. In the technical specification TC5 [28], safety-related function blocks are specified at a high level while the actual implementation is left to the application developer. The *emergency stop* function block [28, pp. 40 – 45], which we consider in the following, is a function block intended for monitoring an emergency stop button.

Interfaces The function block senses five Boolean inputs, namely *Activate*, *S_EStopIn*, *S_StartReset*, *S_AutoReset*, *Reset* and drives three boolean outputs *Ready*, *S_EStopOut*, *Error* and one 16-bit wide diagnosis output *DiagCode*. *S_EStopOut* is the output for the safety-related response.

Requirements The functional description of the block is given by PLCopen as a state diagram [28, p. 42]; Figure 3 shows a simplified version of the state machine (transition conditions and transitions from any state to the idle state (S_1) have been omitted to make the presentation accessible). Overall, the block comprises nine states, i.e., *Idle* (S_1), *Init* (S_2), *Wait for S_EStopIn1* (S_3), *Wait for Reset 1* (S_4), *Reset Error 1* (S_5), *Safety Output Enabled* (S_6), *Wait for S_EStopIn2* (S_7), *Wait for Reset 2* (S_8), and *Reset Error 2* (S_9).

Implementation The implementation consists of approximately 150 lines of low level C code targeting the Intel MCS-51 microcontroller. For our experiments, we used the Keil μVision3 compiler. The compiled and linked `*.hex` file is written into the Intel MCS-51's PROM, serving as the system under test.

4.1 ptLTL Specification

In the implementation, the 8-bit unsigned variable *currState* represents the current state, of the function block. An enumeration maps the state numbers $\{S_1, \ldots, S_9\}$ to identifiers. To simplify presentation, we write Θ_{Sx} as abbreviation for the event $currState = S_x$. We proceed by describing two desired properties of the system.

Property 1 Predecessors of state *Safety Output Enabled* (S_6) are $\{S_2, S_4, S_7, S_8\}$, thus, S_6 shall not be reached from any other state, which is formalized as:

$$\psi^1 := \uparrow(\Theta_{S6}) \rightarrow \big[\uparrow(\Theta_{S2} \vee \Theta_{S4} \vee \Theta_{S7} \vee \Theta_{S8}),\ \uparrow(\Theta_{S1} \vee \Theta_{S3} \vee \Theta_{S5} \vee \Theta_{S9})\big)_S$$

The start of event Θ_{S6} implies that the start of $\{\Theta_{S2}, \Theta_{S4}, \Theta_{S7}, \Theta_{S8}\}$ was observed in the past; since then, the start of $\{\Theta_{S1}, \Theta_{S3}, \Theta_{S5}, \Theta_{S9}\}$ was never observed.

Property 2 Transitions to the reset states *Reset Error 1* (S_5) and *Reset Error 2* (S_9) shall only originate from *Wait for Reset 1* (S_4) and *Wait for Reset 2* (S_8), thus, have only a single predecessor state.

$$\psi^2 := \uparrow(\Theta_{S5}) \rightarrow \downarrow(\Theta_{S4})$$
$$\psi^3 := \uparrow(\Theta_{S9}) \rightarrow \downarrow(\Theta_{S8})$$

The start of event Θ_{S5} causes the end of event Θ_{S4}; the start of Θ_{S9} causes the end of Θ_{S8}.

4.2 Online Runtime Verification

We synthesized observers for properties ψ^1, ψ^2, and ψ^3 (cf. Fig. 5), both as VHDL hardware description and Java code. We sampled the emergency stop module with different, randomly-generated input patterns and could not find a property violation. To prove our approach feasible, we intentionally altered the next-state code of the state-machine implementation in a way that the transition from S_7 to S_8 is replaced by a transition from S_7 to S_9, thus conflicting with ψ^3.

Error scenario The relevant C code of the implementation is listed in Fig. 4. Whereas the code on the left shows the correct implementation of state *Wait for S_EStopIn1* (S_3), the code on the right erroneously introduces a transition to the state *Reset Error 2* (S_9). We first synthesize hardware observers for ψ^1, ψ^2, and ψ^3. Next, the host application configures the atomic checker unit with the atomic propositions that need to be evaluated, that is:

$$ap_1: \quad \Theta_{S8} \triangleq (\text{currState} = \texttt{ST\_WAIT\_FOR\_RST2})$$
$$ap_2: \quad \Theta_{S9} \triangleq (\text{currState} = \texttt{ST\_RST\_ERR2})$$

The (Boolean) verdicts over the atomics are the inputs to the synthesized ptLTL observer, i.e., the vector `atomics` of the VHDL entity shown in Fig. 5. The Boolean

```
case ST_WAIT_FOR_ESTOPIn2:
  Ready = true;
  S_EStopOut = false;
  Error = false;
  DiagCode = 0x8004;
  if(!Activate)
    currState = ST_IDLE;
  if(S_EStopIn && !S_AutoReset)
    currState = ST_WAIT_FOR_RST2;
  if(S_EStopIn && S_AutoReset)
    currState = ST_SAFETY_OUTP_EN;
break;
```

```
case ST_WAIT_FOR_ESTOPIn2:
  Ready = true;
  S_EStopOut = false;
  Error = false;
  DiagCode = 0x8004;
  if(!Activate)
    currState = ST_IDLE;
  if(S_EStopIn && !S_AutoReset)
    currState = ST_WAIT_FOR_RST2;
  if(S_EStopIn && S_AutoReset)
    currState = ST_RST_ERR2;
break;
```

Fig. 4. Emergency stop C implementation; correct(left) and erroneous (right)

output `err` is raised to `true` whenever the specification is falsified by the monitor. The sequential process `p_reset` takes care of initialization of the involved registers and the combinatorial process `p_observer_logic` implements the actual observer for ψ^3. We again applied a random input pattern and revealed the erroneous state transition. For example, the sequence $S_1 \succ S_2 \succ S_6 \succ S_7 \succ S_9 \succ S_6$ was shown (by the observer) to be conflicting with specification ψ^3.

4.3 Offline Runtime Verification

To conclude the example, we also applied our offline approach to the *emergency stop* example. We thus synthesized a Java class serving as monitor and used the event logger of the hardware monitor unit to offer state updates δ to the host. Likewise, the host application was also able to reveal the erroneous state transition. However, offline runtime verification requires a host computer to be present, whereas our online approach is a self-contained hardware approach.

5 Related Work

As our approach supports software as well as hardware-based monitoring functionality, we categorize related work into software- and hardware-based approaches.

Software-based monitoring The commercial tool Temporal Rover [8] allows to check future and past time temporal formulae using instrumentation of source code. Basically, the tool is a code generator that supports Java, C, C++, Verilog or VHDL; properties to be checked are embedded in the comments of the source code. The respective property checks are then automatically inserted into the code, compiled, and executed.

Academic tools with automated code instrumentation capabilities are the Java PathExplorer (JPaX) [13], the Monitoring and Checking (MaC) framework [20], and the Requirements Monitoring and Recovery (Rmor) [14] tool. JPaX and MaC facilitate automated instrumentation of Java bytecode; upon execution, they send a sequence of events to an observer. JPaX additionally supports

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity FORMULA_PSI_3 is
5    generic (
6      ATOMICS_LEN : positive := 2;
7      SUBFORMULAS_LEN : positive := 5);
8    port (
9      clk     : in  std_logic;
10     reset   : in  std_logic;
11     atomics : in  std_logic_vector (ATOMICS_LEN-1 downto 0);
12     err     : out std_logic);
13 end FORMULA_PSI_3;
14
15 architecture behaviour of FORMULA_PSI_3 is
16   signal pre_reg, pre_reg_next  :  std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
17   signal now_reg, now_reg_next  :  std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
18   signal atomics_reg            :  std_logic_vector (ATOMICS_LEN-1 downto 0);
19
20 begin
21
22   p_observer_logic : process(pre_reg, now_reg, atomics_reg)
23     variable pre_reg_next_v : std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
24     variable now_reg_next_v : std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
25   begin
26     pre_reg_next_v := pre_reg;
27     now_reg_next_v := now_reg;
28
29     now_reg_next_v(4) := atomics_reg(0);
30     now_reg_next_v(3) := not now_reg_next_v(4) and pre_reg_next_v(4);
31     now_reg_next_v(2) := atomics_reg(1);
32     now_reg_next_v(1) := now_reg_next_v(2) and not pre_reg_next_v(2);
33     now_reg_next_v(0) := not now_reg_next_v(1) or now_reg_next_v(3);
34     pre_reg_next_v    := now_reg_next_v;
35
36     pre_reg_next  <= pre_reg_next_v;
37     now_reg_next  <= now_reg_next_v;
38   end process;
39
40   p_reset : process (clk, reset)
41     variable pre_reg_v : std_logic_vector (SUBFORMULAS_LEN-1 downto 0);
42   begin
43     if reset = '1' then
44       pre_reg_v(4) := atomics(0);
45       pre_reg_v(3) := '0';
46       pre_reg_v(2) := atomics(1);
47       pre_reg_v(1) := '0';
48       pre_reg_v(0) := not pre_reg_v(1) or pre_reg_v(3);
49       pre_reg      <= pre_reg_v;
50       now_reg      <= (others => '0');
51       atomics_reg  <= (others => '0');
52     elsif rising_edge(clk) then
53       pre_reg      <= pre_reg_next;
54       now_reg      <= now_reg_next;
55       atomics_reg  <= atomics;
56     end if;
57   end process;
58
59   err <= not now_reg(0);
60
61 end behaviour;
```

Fig. 5. The auto-generated observer VHDL code for ψ^3

concurrency analysis. RMOR provides a natural textual programming notation for state machines for program monitoring and implements runtime verification for C code.

Hardware-based monitoring Tsai et al. [33] describe a noninterference hardware module based on the MC68000 processor for program execution monitoring and data collection. Events to be monitored, such as function calls, process creation, synchronization, etc. , are predetermined. With the support of a replay controller, test engineers can replay the execution history of the erroneous program in order to determine the origin of the defect. The Dynamic Implementation Verification Architecture (DIVA) exploits runtime verification at intra-processor level [1]. Whenever a DIVA-based microprocessor executes an instruction, the operands and the results are sent to a checker which verifies correctness of the computation; the checker also supports fixing an erroneous operation. A hardware-related tool called BUSMOP [27] is based on the Monitor Oriented Programming (MOP) framework [6]. In essence, BUSMOP is a hardware-monitoring device which *sniffs* traffic transmitted between COTS embedded components attached to a PCI/PCI-X bus, thereby acting as *advanced* bus guardian. Similar to our approach, the monitor and the system under verification are executed within an FPGA. The specification is translated by the MOP framework into a hardware description, which is then synthesized into a netlist and loaded into dynamically reconfigurable blocks of the FPGA. Whereas BUSMOP is designed to monitor data transmissions through a PCI interconnection for large-scale embedded systems, our framework monitors embedded software at a fine level of granularity.

The work of Brörkens and Möller [5] is akin to ours in the sense that they also do not rely on code instrumentation to generate event sequences. Their framework, however, targets Java and connects to the bytecode using the Java Debug Interface (JDI) so as to generate sequences of events.

Lu and Forin [25] present a compiler from Property Specification Language (PSL) to VERILOG, which translates a subset of PSL assertions about a software program (C in their approach) into hardware execution blocks for an extensible MIPS processor, thus being the first method that allows transparent runtime verification without altering the program under investigation. The synthesized verification unit is generated by a property rewriting algorithm proposed in [32]. Atomic propositions are restricted to allow only a single comparison operator, whereas our approach supports more complex relations among memory values within our hardware unit, thus yielding greater flexibility in the specification.

Observer synthesis The idea of generating Java code as observers for ptLTL is due to Havelund and Roşu [15]. A comparable approach based on alternating automata for future time LTL was described by Finkbeiner and Sipma [11].

6 Conclusion and Future Challenges

This paper advocates runtime verification of microcontroller code without code instrumentation. Our method supports runtime checks for ptLTL during execution of the code, thereby evading the problem of errors introduced by translation from a high-level language into binary code. Such errors are likely to go unnoticed by conventional approaches for high-level representations. The framework itself relies on a hardware monitor unit and synthesized observers, thereby making code instrumentation dispensable. The example discussed in this paper is based on randomly generated inputs, which is insufficient in practical applications. Test-case generation for binary code, though orthogonal to the techniques described in this paper, thus remains a topic of interest. For this task, we will further investigate a combination of SAT solving and backward abstract interpretation [4, 30].

Acknowledgement 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 by the DFG research training group 1298 *Algorithmic Synthesis of Reactive and Discrete-Continuous Systems*.

References

1. Austin, T.M.: DIVA: A reliable substrate for deep submicron microarchitecture design. In: MICRO. pp. 196–207. IEEE (1999)
2. Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T.: WYSINWYX: What you see is not what you execute. In: VSTTE. Toronto, Canada (2005)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from unstructured programs. In: VMCAI (2011), to appear
4. Brauer, J., King, A.: Transfer function synthesis without quantifier elimination. In: ESOP. LNCS, vol. 6602, pp. 97–115. Springer (2011)
5. Brörkens, M., Möller, M.: Dynamic event generation for runtime checking using the JDI. Electronic Notes in Theoretical Computer Science 70(4), 21 – 35 (2002)
6. Chen, F., Roşu, G.: MOP: An efficient and generic runtime verification framework. In: OOPSLA. pp. 569–588. ACM (2007)
7. Colin, S., Mariani, L.: Model-Based Testing of Reactive Systems, chap. Run-Time Verification, pp. 525–555. Springer (2005)
8. Drusinsky, D.: The temporal rover and the ATG rover. In: 7th Intl. SPIN Workshop on SPIN Model Checking and Software Verification. pp. 323–330. Springer (2000)
9. Eide, E., Regehr, J.: Volatiles are miscompiled, and what to do about it. In: EMSOFT. pp. 255–264. ACM (2008)
10. Emerson, E.A.: Handbook of theoretical computer science (vol. B), chap. Temporal and modal logic, pp. 995–1072. MIT Press (1990)
11. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Form. Methods Syst. Des. 24, 101–127 (March 2004)
12. Flexeder, A., Mihaila, B., Petter, M., Seidl, H.: Interprocedural control flow reconstruction. In: APLAS. LNCS, vol. 6461, pp. 188–203. Springer (2010)

13. Havelund, K., Roşu, G.: An overview of the runtime verification tool Java PathExplorer. Form. Methods Syst. Des. 24(2), 189–215 (2004)
14. Havelund, K.: Runtime verification of C programs. In: TestCom/FATES. pp. 7–22. Springer (2008)
15. Havelund, K., Rosu, G.: Synthesizing monitors for safety properties. In: TACAS. pp. 342–356. LNCS, Springer (2002)
16. Howe, J., King, A.: Logahedra: A new weakly relational domain. In: ATVA, LNCS, vol. 5799, pp. 306–320. Springer (2009)
17. Kinder, J., Veith, H., Zuleger, F.: An abstract interpretation-based framework for control flow reconstruction from binaries. In: VMCAI. LNCS, vol. 5403, pp. 214–228. Springer (2009)
18. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer (2008)
19. Laroussinie, F., Markey, N., Schnoebelen, P.: Temporal logic with forgettable past. In: LICS. pp. 383–392. IEEE (2002)
20. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: PDPTA. pp. 279–287 (1999)
21. Leroy, X.: Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: POPL. pp. 42–54. ACM (2006)
22. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. 43, 363–446 (December 2009)
23. Lichtenstein, O., Pnueli, A., Zuck, L.: The glory of the past. In: Logics of Programs, LNCS, vol. 193, pp. 196–218. Springer (1985)
24. Lindig, C.: Random testing of C calling conventions. In: AADEBUG. pp. 3–12. ACM, New York, NY, USA (2005)
25. Lu, H., Forin, A.: The design and implementation of P2V, an architecture for zero-overhead online verification of software programs. Tech. Rep. MSR-TR-2007-99, Microsoft Research (2007)
26. Parr, T.J., Quong, R.W.: ANTLR: a predicated-ll(k) parser generator. Softw. Pract. Exper. 25, 789–810 (1995)
27. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. Real-Time Systems Symposium pp. 481–491 (2008)
28. PLCopen: Safety software, technical specification, Part 1: Concepts and function blocks. online (2006)
29. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: TACAS. LNCS, vol. 1384, pp. 151–166. Springer (1998)
30. Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., Kowalewski, S.: Test-case generation for embedded binary code using abstract interpretation. In: MEMICS. pp. 151–158 (2010)
31. Reinbacher, T., Horauer, M., Schlich, B., Brauer, J., Scheuer, F.: Model checking assembly code of an industrial knitting machine. In: EM-Com. pp. 97–104. IEEE (2009)
32. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automated Software Eng. 12(2), 151–197 (2005)
33. Tsai, J.J.P., Fang, K.Y., Chen, H.Y., Bi, Y.: A noninterference monitoring and replay mechanism for real-time software testing and debugging. IEEE Trans. Softw. Eng. 16, 897–916 (1990)