Automated Test-Trace Inspection for Microcontroller Binary Code

Thomas Reinbacher[1], Jörg Brauer[2], Daniel Schachinger[1], 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

Abstract. This paper presents a non-intrusive framework for runtime verification of executable microcontroller code. A dedicated hardware unit is attached to a microcontroller, which executes the program under scrutiny, to track atomic propositions stated as assertions over program variables. The truth verdicts over the assertions are the inputs to a custom-designed μCPU unit that evaluates past-time LTL specifications in parallel to program execution. To achieve this, the instruction set of the μCPU is tailored to determining satisfaction of specifications.

1 Introduction

Real software runs on real machines. Ideally, verification should thus take place on the execution level. A main advantage of this approach is that it eliminates the need for compiler correctness, which is extremely difficult to establish. However, analyzing programs on the machine-level poses other challenges, even more so in the embedded systems domain where there is heavy interaction between the software and its environment. As a consequence, in practice, only certain parts of the program may be backed up with a formal correctness argument. For the remaining part of the program, testing is often the technique of choice to increase confidence in correctness of the program without proving absence of errors.

Testing is based on a guess-and-check paradigm: one (a) *guesses* a configuration of the program's inputs (the test-case) and (b) *checks* the result of the individual test runs. While the former can — to a large extent — be automated by automated test-case generation [1], the latter often turns out to be a time-consuming and manual activity, remaining a core task of test engineers. With respect to test automation, it is therefore highly desirable to automatically evaluate the validity of a single test trace when running in the intended execution environment. Runtime verification further ties verification to testing: The intended behavior of the system is described in some suitable temporal logic formula, the validity of which is monitored dynamically, while the test case is executed. Yet, in the context of safety-critical embedded systems, the application of runtime verification on execution level is hampered by the fact that code instrumentation — which is required by traditional techniques — is likely to affect certain real-time and memory constraints of the system. This is specifically serious in applications where the design tightly fits into the available resources.

In previous work [8], we synthesized VHDL code representing a monitor for a past-time LTL [4] (ptLTL) formulae. The truth values of the atomic propositions (APs) as well as the validity of the specification were evaluated in a pure hardware solution. The approach proves feasible in a *static* setting, where one checks a fixed set of properties at every run of the program, e.g., after the product is shipped. However, in a *dynamic* setting such as testing, the specification is likely to change with every single test execution. Since generating a hardware observer from VHDL requires invoking a logic synthesis tool (which may take several minutes), this approach is infeasible for testing. To make runtime verification amenable to real-world testing, this paper proposes a more general approach that relies on a μCPU to determine satisfaction of ptLTL properties on-the-fly. APs are (still) evaluated by a dedicated, configurable hardware unit.

2 Runtime Verification for Microcontroller Binary Code

This section presents our framework for non-intrusive runtime verification of microcontroller binary code (see Fig. 1). APs are evaluated in a component called the atChecker, whereas satisfaction of a ptLTL formula is determined by a μCPU unit, the μMonitor. A control unit wiretaps the memory of the microcontroller that executes the software under investigation. To illustrate our (mostly generic) approach, we employ an off-the-shelf Intel MCS-51 microcontroller IP-core for our experiments. Since verification is performed on the binary program, this approach does not impose any constraints on the high-level implementation language.

Specification Our framework supports specifications in ptLTL augmented with monitoring operators [7]. A GUI-based host application compiles a specification (consisting of a set of formulae) into a pair $\langle \Pi, \mathcal{C} \rangle$, where $\mathcal{C}$ is a configuration for the atChecker and Π is a set of native programs for the μMonitor. To do so, we instantiate an algorithm proposed by Havelund and Roşu [7] to generate observers for ptLTL. If available, we parse debug information generated during compilation to relate program symbols to memory locations on the microcontroller. This allows us to use high-level program symbols in specifications, for example, $\psi : \uparrow (\mathsf{foo} = 20) \Rightarrow \mathsf{bar} \leq 50$; where foo and bar are variables.

Evaluating Assertions On-The-Fly The atChecker supports a subclass of two-variable inequalities, namely those of the form $\alpha \cdot m_1 + \beta \cdot m_2 \bowtie C$ where $\alpha, \beta \in \{0, \pm 2^n \mid n \in \mathbb{N}\}$, m_1, m_2 are locations within RAM, $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$, and $C \in \mathbb{Z}$ is a constant. These assertions are easily evaluated in hardware using shifters and adders. One unit is used for each AP of the specification.

Evaluating ptLTL specifications The μMonitor is a non-pipelined, RISC-based microcomputer featuring an instruction set that supports sequential evaluation of ptLTL specifications. It has separate address spaces for program and data memory, i.e., represents a Harvard architecture. The data memory consists

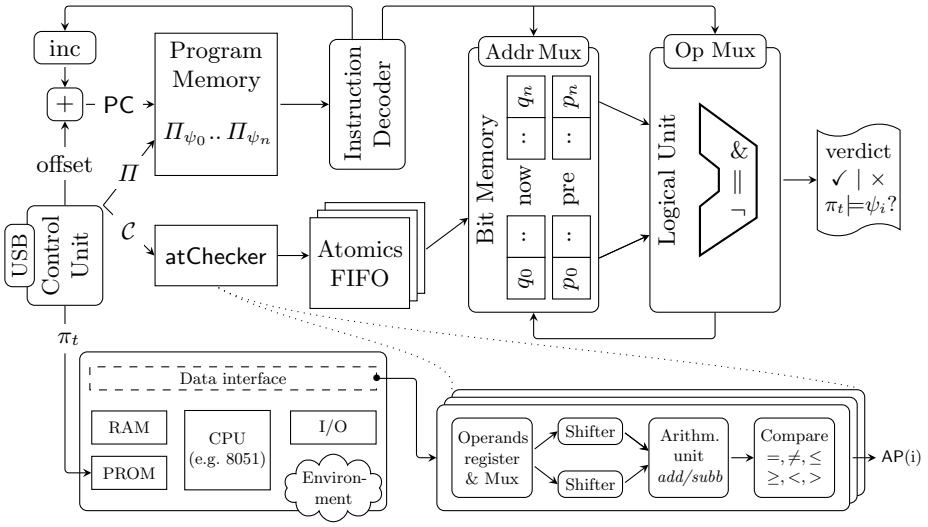

Fig. 1. μMonitor (top), atChecker (bottom right), and the SUT (bottom left)

of two registers, one holding the evaluations (true, false) of all subformulae of the formula ψ in the current execution cycle $q[0 \dots n]$ and one the results of the previous cycle $p[0 \dots n]$. All bits in the data memory are directly addressable. The program memory, in turn, is partitioned into n sections, each holding a program $\pi_\psi \in \Pi$ compiled from ψ. The host computer selects an individual program by setting an offset that is added to the current program counter. This easily allows to change the specification on-the-fly, e.g., whenever a new test-case is loaded.

Each program π_ψ is executed in cycles. A cycle starts with the first address belonging to π_ψ and ends when the last instruction was executed. At the end of a cycle, the verdict is updated to indicate whether ψ holds up to the current state of the program. The start of a cycle is triggered whenever any of the APs change their truth values. To illustrate, consider again $\psi : \uparrow (\mathsf{foo} = 20) \Rightarrow \mathsf{bar} \leq 50$. A cycle of π_ψ is triggered iff $[\mathsf{foo} = 20]$ or $[\mathsf{bar} \leq 50]$ toggle their truth values.

The *instruction set* features 16 opcodes to handle the ptLTL operators, where each opcode is three bytes long. An instruction decoder allows to address individual bits in the data memory and set the operator for the *logical unit*. A multi-way multiplexer (the logical unit) connects bits, originating from either $p[0 \dots n]$ or $q[0 \dots n]$, with a Boolean operator $op \in \{\neg, \wedge, \vee\}$ and transfers the result back to memory. The whole framework results in an efficient hardware design. The μMonitor unit synthesizes down to 367 logic cells (with $f_{max} = 145\,\text{MHz}$) and a single atChecker unit to 290 logic cells (with $f_{max} = 80\,\text{MHz}$) on an Altera Cyclone III EP3C16 FPGA device. By way of comparison, the Intel MCS-51 core consumes roughly 4000 logic cells on the same device and runs at clock speed of up to 16 MHz.

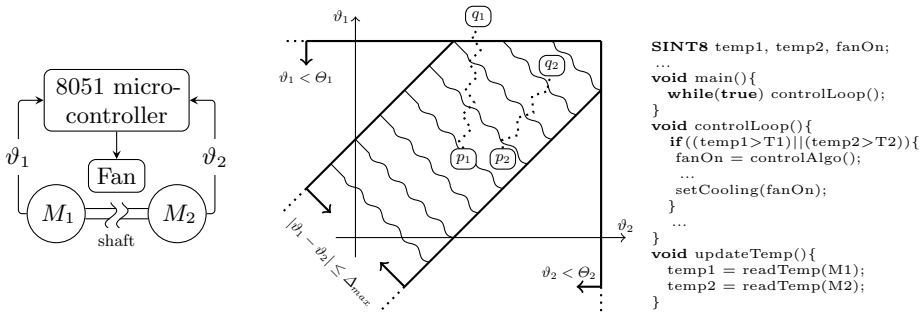

Fig. 2. Application (left), specification (mid), and the source code (right)

3 Worked Example

To exemplify our framework, we turn to an application in industrial automation with the following specification: "The program under scrutiny is a digital controller implementation controlling the temperature of two DC motors M_1 and M_2 by driving a fan. The motors have a maximum operating temperature Θ_1 and Θ_2, respectively. The target application continuously reads the current operating temperatures ϑ_1 and ϑ_2. The applications invokes cooling whenever either $\vartheta_1 > T_{on_1}$ or $\vartheta_2 > T_{on_2}$. To avoid damage of the motors along with functional deficiency, the fan needs to be turned on before the temperature of the motors reaches their critical temperature. Both motors operate on the same shaft, thus, an additional sanity check is that the absolute temperature difference $|\vartheta_1 - \vartheta_2|$ remains within Δ_{max}, otherwise, we could assume that one of the motors is blocking while the other needs to apply an unusually high torque."

The implementation consists of approx. 250 lines of C (compiled with Keil μVision3). An outline of the code structure is shown in Fig. 2 (right). The function updateTemp() is periodically called from a timer interrupt, whereas controlAlgorithm() holds the controller implementation. Intuitively, the different temperature bounds describe four hyper-planes as shown in Fig. 2 (mid). Consider the temperature pattern from p_1 to q_1. The controlLoop turns on the cooling in p_1 after one of the thresholds is reached. After returning from controlAlgorithm() and turning on the fan, the temperatures are already at q_1, violating the temperature requirement of M_1. However, the pattern from p_2 to q_2 is valid wrt. the specification as the temperature curve never leaves the hatched area until the fan is turned on. It is thus straightforward to come up with the specification:

$$\psi : \mathsf{Inv}(|\vartheta_1 - \vartheta_2| \leq \Delta_{max}) \quad \bigwedge$$
$$\uparrow (\mathsf{fanOn} = \#\mathsf{F\_ON}) \Rightarrow [\vartheta_1 > T_{on_1} \vee \vartheta_2 > T_{on_2}\ ;\ \vartheta_1 \geq \Theta_1 \vee \vartheta_2 \geq \Theta_2)_s$$

The symbols ϑ_1 and ϑ_2 in ψ refer to the variables temp1 and temp2. ψ requires that: (a) The absolute temperature difference between M_1 and M_2 shall never

be greater than Δ_{max} and (b) whenever the fan is turned on then one of the motor temperatures exceeded its threshold in the past, and since then none of the temperatures exceeded its critical temperature. Inv stands for *invariant*, i.e., holds in every state, $\uparrow$ means *rising* (false in the previous state but true in the current), and $[p; q)_s$ is the *strong interval operator* [7] (q was never true since the last time p was observed to be true, including the state when p was true). The bounds are set to $\Delta_{max} = 40°C$, $T_{on_1} = 30°C$, $T_{on_2} = 35°C$, $\Theta_1 = 100°C$, and $\Theta_2 = 90°C$. For ψ, the host application generates a program consisting of 13 instructions for the μMonitor and a configuration to evaluate the 7 APs of ψ for the atChecker. The application as well as the monitor execute at full clock rate.

4 Concluding Discussion

This paper presents a custom-designed μCPU unit for non-intrusive runtime monitoring of ptLTL. The μCPU as well as hardware circuits for checking APs are wiretapped to an FPGA running the target hardware. The force of this approach is that the μCPU can be reprogrammed dynamically, depending on the specification to be checked, whereas previous approaches evaluated formulae using fixed hardware circuits, which is clearly not as flexible. In contrast to software-based solutions such as Temporal Rover [3], JPaX [5], or Rmor [6], our framework does not require instrumentation. Existing hardware-based approaches [2, 9] require sophisticated monitoring devices, whereas our framework simply wiretaps the microcontroller's memory on an FPGA. Future work will be the integration of our framework with binary code analysis frameworks that generate the actual test cases, rather than using randomly generated executions as done currently.

References

1. Belinfante, A., Frantzen, L., Schallhart, C.: Tools for test case generation. In: Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 67–76. Springer (2005)
2. Chen, F., Roşu, G.: MOP: An efficient and generic runtime verification framework. In: OOPSLA. pp. 569–588. ACM (2007)
3. Drusinsky, D.: The temporal rover and the ATG rover. In: SPIN. pp. 323–330. Springer (2000)
4. Emerson, E.A.: Handbook of theoretical computer science (vol. B), chap. Temporal and modal logic, pp. 995–1072. MIT Press (1990)
5. Havelund, K., Roşu, G.: An overview of the runtime verification tool Java PathExplorer. Form. Methods Syst. Des. 24(2), 189–215 (2004)
6. Havelund, K.: Runtime verification of C programs. In: TestCom/FATES. pp. 7–22. Springer (2008)
7. Havelund, K., Roşu, G.: Synthesizing monitors for safety properties. In: TACAS. pp. 342–356. LNCS, Springer (2002)
8. Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., Kowalewski, S.: Past time LTL runtime verification for microcontroller binary code. In: FMICS (2011)
9. 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)