Science of Computer Programming 80 (2014) 109–129

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Runtime verification of microcontroller binary code

Thomas Reinbacher [a,*], Jörg Brauer [b,c], Martin Horauer [d], Andreas Steininger [a], Stefan Kowalewski [c]

[a] Embedded Computing Systems Group, Vienna University of Technology, Austria
[b] Verified Systems International GmbH, Bremen, Germany
[c] Embedded Software Laboratory, RWTH Aachen University, Germany
[d] Department of Embedded Systems, University of Applied Sciences Technikum Wien, Austria

ARTICLE INFO

Article history:
Received 30 January 2012
Received in revised form 10 July 2012
Accepted 31 October 2012
Available online 21 November 2012

Keywords:
Runtime verification
Past time LTL
Embedded real-time systems

ABSTRACT

Runtime verification bridges the gap between formal verification and testing by providing techniques and tools that connect executions of a software to its specification without trying to prove the absence of errors. This article presents a framework for runtime verification of microcontroller binary code, which provides the above mentioned link in a non-intrusive fashion: the framework neither requires code instrumentation nor does it affect the execution of the analyzed program. This is achieved using a dedicated hardware unit that runs on a field programmable gate array in parallel to the analyzed microcontroller program. Different instances of this framework are discussed, with varying degrees of expressiveness of the supported specification languages and complexity in the hardware design. These instances range from invariant checkers for a restricted class of linear template constraints to a programmable processor that supports past-time linear temporal logic with timing constraints.

1. Introduction

Long-standing research efforts in the fields of programming languages, compilers, and computer security have led to the development of a large number of techniques to analyze programs for errors, security vulnerabilities, and all different kinds of runtime properties. These techniques include, most notably, model checking, static analysis, and theorem proving, which have been implemented in a wide variety of tools [1–17]. Most of the tools deal with the problem of proving that all possible executions of a program adhere to its specification – which is typically given in some formal logic – whereas others serve as bug-hunting frameworks. Yet, despite the impressive technical progress and the increasing number of success stories from industry, this is an extremely challenging task due to the complexity as well as the sheer size of modern software systems. Indeed, proving the functional correctness of a system consisting of about ten thousand lines of code represents the state-of-the-art [18]. Most real-world software clearly exceeds these limitations.

1.1. Fault detection by testing

Conventional testing is less ambitious. Rather than proving that all possible executions of a system satisfy its specification, only a finite subset of these executions is examined during the (software) development process. Hence, testing merely raises confidence in the correctness (or reliability) of the system, instead of proving the absence of runtime errors [19]. Industry

* Corresponding author.
E-mail address: treinbacher@ecs.tuwien.ac.at (T. Reinbacher).

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.10.015

typically enforces a strict development process where testing is the predominant method, even if highest correctness requirements apply, e.g., because of ultra-high reliability requirements which are present in the avionics industry [20]. One may argue that this is due to the lack of feasible (and affordable) alternatives. The classical approach in testing follows a guess-and-check paradigm: one guesses a configuration of the program inputs (the test case) and checks the results of the individual test runs. The former can – to a large extent – be automated by test-case generation techniques [21], which often follow some coverage criterion. However, the arduous task of defining the reference results of each test-case execution then typically remains with a test engineer. With respect to test automation, it is thus highly desirable to automatically evaluate the outcome of a single test trace when executed.

1.2. Runtime verification by code instrumentation

In recent years, runtime verification [22] has gained increasing interest as it bridges the gap between traditional formal verification and testing. As in testing, runtime verification does not prove the absence of errors, yet it provides a systematic way to link the test-case executions to a formal specification. 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 a monitor [23,24] which evaluates sequences of events in an instrumented version of the program under scrutiny. Instrumentation can either be done manually or automatically, for instance, by scanning available programs for assignments and function calls at the level of the implementation language. Function calls are then inserted to emit relevant events to an observer. 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 or Verilog. An important advantage of runtime verification over testing is that runtime verification allows the monitor to respond to the analyzed system, a technique that is frequently referred to as *steering*. For example, if a runtime verification framework detects a serious flaw in a railway control system that is in operation, it can steer the system into a *safe state*, thereby avoiding failure [25]. Various runtime verification frameworks have thus emerged [26–30].

1.3. Pitfalls of code instrumentation

Yet, despite considerable technical progress, existing approaches to runtime verification are not directly transferable to the domain of (safety-critical) embedded systems software, mainly due to the following reasons:

- Code instrumentation increases memory consumption. This factor is of economical relevance for small-sized embedded target platforms and applies to both, the program memory as well as RAM. The relevance of this aspect is emphasized by the fact that restricted architectures are often used in critical environments [31] (such as nuclear power plants) to avoid unpredictable timing behavior, e.g., caused by caches.
- The timing behavior of the system is altered by code instrumentation [32,33]. The additional runtime overhead may have devastating consequences for heavy-loaded real-time applications with tight deadlines.
- Code instrumentation alters the program and is thus impracticable for certified systems as it requires the re-certification of the affected program parts.
- Embedded code often adopts target-specific language extensions, direct hardware register and peripheral access, and embedded assembly code [34,35]. When instrumenting a high-level code basis that uses such features, one has to take all the particularities of the target hardware into account, depleting the prospect of a generic approach.
- Instrumentation at binary code level, in turn, is incomplete (i.e., it does not provide 100% instrumentation coverage) as long as a sound approximation of the control flow graph (CFG) is not reconstructed from the binary program.[1] Although CFG reconstruction from executable code is an active area of research [38–41], generating sound yet precise results remains a challenge.
- In its present shape, runtime verification analyzes the correctness of high-level code. However, to show that a high-level specification is correctly reproduced by the executable program, it is further necessary to verify the translation applied to the high-level code (as it is not unknown for compilation to introduce errors [42–44]). One thus needs to prove that for a given source program P, if the compiler generates a binary code B without errors, then B behaves like P [45]. Formally verified compilers are a topic in research for almost half a century [46,47], however, despite some recent breakthroughs [48,49], only few verified compilers are used in practice. Flaws introduced by a compiler may thus remain undetected by existing approaches.

In a recent paper, Pike et al. [50, Section 3] have studied the application of runtime verification to systems with ultra-high reliability requirements. There, they have provided a generalization of these low-level aspects, called *FaCTS*:

- Functionality: The runtime verification system must not change the target's behavior.
- Certifiability: The runtime verification system must not make re-certification of the target onerous.

[1] Of course, this only applies to static instrumentation techniques. Dynamic binary instrumentation frameworks such as Valgrind [36,37] use dynamic techniques akin to just-in-time compilation, which sidesteps this problem.