Efficient System-Level Testing of Embedded Real-Time Software

Daniel Sundmark[†], Anders Pettersson[†], Sigrid Eldh[†‡], Mathias Ekman[†⋆], and Henrik Thane[†]
[†]MRTC, Mälardalen University, Box 883, SE-721 23 Västerås, Sweden
[⋆]Bombardier Transportation, Dept. PPC/ETS, SE-721 73, Västerås, Sweden
[‡]Ericsson, Älvsjö, Sweden
Daniel.Sundmark@mdh.se

Abstract

When developing complex embedded applications, the software is often built in the form of subsystems or components, which are later integrated and assembled to a full system. Throughout all stages of development and assembly, software testing is performed. On unit- or component-level, the structure and run-time properties of the software can fairly easily be predicted. This enables a thorough testing based on the structure of the software (as well as testing based on the intended functionality of the software). However, in the later stages of the assembly process, the structure of the software becomes more complex and less obvious. Hence, the developers are forced to base their testing solely on the intended functionality of the software, leading to a reduced testing-induced quality assurance. At the same time, bugs found in the later stages of testing (i.e., integration- and system-level testing) are significantly more costly to correct. Using dynamic and static information of the software, this situation can be significantly improved. This paper outlines a framework for more efficient system-level testing of real-time software for embedded applications.

1 Introduction

Premium quality software is coveted by all, but (using today's state-of-the-practice tools) attainable by few. Depending on the means of measurement, the costs of software testing, debugging, and regression testing range between 30–90% of the overall product life-cycle cost in software development [8]. There is no simple remedy to this situation, but a lot can be done by performing testing, debugging, and system design in a more structured manner.

Since testing is performed on several different levels (e.g., unit level, integration level, and system level) during software development, there is an abundance of methods for different flavors of testing. Most techniques can, however, be categorized either as structural techniques (where test cases are derived based on the structure of the software) or funtional techniques (where test cases are derived primarily based on the intended functionality of the software). Traditionally, unit level testing is performed using both structural and functional techniques while integration and system level testing are mainly performed using functional techniques. This is mainly due to the difficulties of modeling the execution behavior of system-level software.

Since structural testing is non-trivial on the higher levels of testing (i.e., integration and system testing), software developers have little or no knowledge of the quality assurance achieved by a certain amount of testing. In practice, this results in a situation where large parts of the system behavior remain untested, while others are over-reduntantly tested. Hence, for most software, functional *and* structural testing are needed on all levels of system verification.

1.1 Coverage

Coverage is an important aspect of thoroughness of the testing. The main goal of testing is to find failures, and coverage is a way to show how well tests has "covered" a special aspect of the system. There is a large amount of different coverage techniques (e.g., control flow coverage, data flow coverage, and indata coverage). Some techniques are pragmatic, while others rely on more formalized definitions and models of the system behavior [19].

1.2 Embedded Real-Time Systems

Real-time systems are most often implemented as a set of tasks, running on top of a real-time operating system. In our work, we support two different task run-time models:

- Event-triggered transactions, where a chain of tasks (a transaction) is triggered by an external event. Within the transaction, tasks are triggered via inter-task synchronization primitives, such as message passing or semaphore operations.
- Strictly periodic time-triggered tasks, where a transaction can be seen as a periodically recurring pattern of time-triggered priority-based scheduled tasks. In general, we assume that systems corresponding to this run-time model execute on a small embedded microcontroller (e.g., a vehicular ECU [7]) without jitter-inducing cache or pipeline support.

Testing (and debugging) of embedded real-time systems are primarily complicated by the following aspects: First, the inherent non-determinism of real-time systems (caused by race conditions, hardware interrupts, and interaction with an external context) has a severe impact on reproducibility and predictability. Second, the strict real-time requirements complicate intrusive software instrumentation [6]. Third, the embedded nature of these systems makes their run-time behavior hard to observe when compared to, e.g., desktop applications.

1.3 Contribution

Before, during, and after testing we aquire significant amounts of information of the static and dynamic properties of the software (the obvious test results set aside). Such information includes source code, execution logs, knowledge of the external system context, etc. When performing testing, we have the possibility of using such information in order to improve on the effectiveness and precision of future testing. In our research, our goal is to develop methods for integration and system level testing that use this information as intelligently as possible (see Figure 1).

This paper outlines a framework (or rather, a set of methods) for using this information in order to achieve a more efficient system-level testing of real-time software for embedded applications. This framework further accentuates and clarifies the importance of predictable, observable, and reproducible software behavior.

2 Analysing Static and Dynamic System Information for Efficient System Testing

In Figure 1, an overview sketch of our testing framework is shown. The framework consists of a set of analysis and test methods aimed at establishing reliable structural coverage criteria for system-level testing (thus reducing the cost and increasing the quality of embedded real-time system testing). The remainder of this section will briefly describe a number of these methods.

2.1 Interleaved Control Flow Test Coverage

Control flow graphs are generally used as abstract models of the internal run-time behavior of software. Traditional control flow graphs display all possible execution selections for a piece of software (e.g., a function or a simple sequential program) and serve as the foundation for several different formal coverage critera (as described in Section 1.1). A major problem of control flow analysis is that most software products do not conform to the structure of a function or a simple sequential program. Issues like pseudo-parallellism, parallellism, and interrupts bring another dimension to control flow analysis, a dimension that often proves too complex to handle.

Embedded real-time systems constitute a resource-constrained subset of concurrent systems with more rigorous control on execution behavior. These factors somewhat reduce the run-time complexity allowing otherwise too-complex methods to be used. In our work, we propose the use of Interleaved Control Flow Analysis (ICFA) in order to reduce developer's lack of knowledge regarding system-level test coverage. The basis for our ICFA is the Interleaved Control Flow Graph (ICFG), a system-level model of execution behavior (as in [16]), but with task-level granularity (as in [1]).

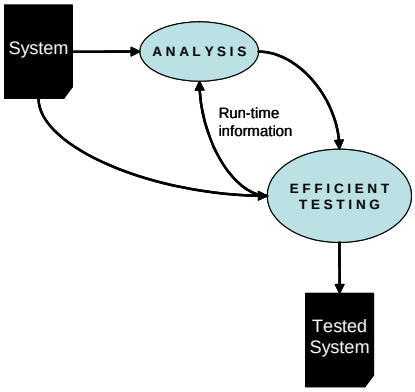

Figure 1. Analysis for efficient embedded real-time system testing.

The ICFG constitutes a solid ground for more structure-based testing on integration- and system level, as well as providing a possibility of performing control flow coverage testing on system-level.

2.2 Regression Test Selection

Regression testing is the process of re-testing test cases following system changes. Looking at software products from a life-cycle perspective, regression testing is required during software maintenance, where functionality is added or bugs are corrected. Basically, whenever software is altered there is a need to verify the alterations, but also to ensure that the changes have not corrupted other parts of the intended functionality of the software. In order to do this, there is a need for a strategy for deciding which test cases to re-test [11]. Ball [1] proposes the use of an intersection graph where automated control flow analysis of the control flow graphs of a program *before* and *after* changes aids in selecting test cases for re-testing.

As many structural testing methods, the intersection graph analysis is constructed to operate on unit-level testing. Within our framework, we intend to adapt this intersection graph analysis on our ICFG:s in order to decide which parts of a multi-tasking real-time system need re-testing following changes (see Figure 2).

2.3 Data Flow Analysis

Test coverage on control flow does not solely cover all aspects of the system behavior. Equally important is the data flow coverage. Data flow coverage is often based on dependencies between definition and usages of variables

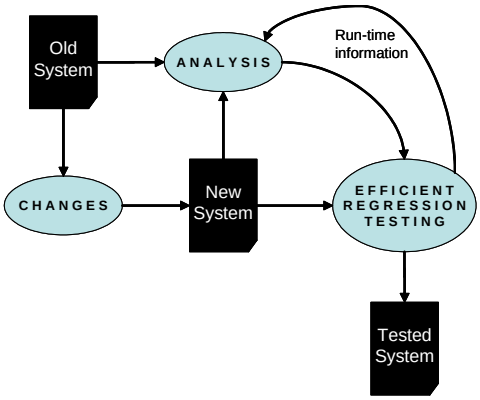

Figure 2. Using knowledge from previous testing round for efficient regression testing.

(i.e., def-use dependencies). Information of the data flow can be derived by using data flow analysis [4, 9].

Another area where the result of the data flow analysis can be used is to reduce the complexity of control flow graphs. By separating possibly valid paths from non-valid paths we are able to prune program-level and system-level control flow graphs from paths describing infeasible software behavior. Examples of such analysis might be the inclusion of initial semaphore values, which will prune several infeasible paths from the graph, lowering the complexity of subsequent analysis.

2.4 Replay-Based Coverage Instrumentation

In order to establish coverage criteria based on task-level control flow, there is a need for observing the control flow path traversed by the execution of the program. In general, such observability requires significant instrumentation of the source code at the basic block level. For sequential software, this kind of instrumentation will not perturb the functional outcome of the tests. However, in real-time systems, the prolonged execution time may significantly affect the non-deterministic constructs of the execution (e.g., race conditions) [6].

We intend to investigate the possibilities of using Deterministic Replay [13] in order to perform low-level coverage instrumentation on replay executions, where race conditions and timing can be deterministically reproduced regardless of intrusive probing. For this purpose, there are also methods available to dynamically add coverage instrumentation code to existing running applications. In particular, a tool-library called DynInst [2] permits the insertion of code into a running application. A mutating application may attach to a running application, create a new bit of code and insert it into the application. There is no need to re-compile, re-link or even re-start the application. By using this approach, the coverage instrumentation can be extended and removed in a deeper level for blocks of interest, without having to prepare the original application at all [17].

3 Related Work

There exist numerous tools and techniques for testing of generel purpose computer software. The majority of these have been developed for solitary (non real-time) sequential programs. As a consequence, they are not applicable to industrial real-time systems, since they disregard issues of timing and concurrency. Also, most of these tools depend on the assumption that the system is developed from scratch and that there is no legacy code to account for.

In multi-tasking execution, where tasks are competing for resources due to concurrent execution, tasks test repeatiblity (reproducibility) is not always guaranteed. Reproducibility is essential for regression testing and cyclic debugging, where the same test cases are run repeatedly with the intention of verifying modified program code or to track down errors [5]. It is common for real-time software to have a non-reproducible behavior. Under such circumstances, iving the same input and same internal state to a program is not sufficient, since race conditions and interrupts may give different execution orderings from the same initial state [12, 18]. An aspect of this non-determinism is intrusive observations caused, e.g., by temporary additions of program code for diagnostics, which incurs a probe-effect [6] by changing the race conditions in the system. We have previously published methods that address the problem of testing concurrent real-time software during the system integration phases [10, 16].

As for execution replay, an early version of our Deterministic Replay technique, which supported replay of interrupts, preemption of tasks and distributed transactions is described in [15]. However, this work assumed the existence of special off-line versions of real-time operating systems (RTOS), which is not a plausible assumption for current commercial real-time operating systems. Recently we have elaborated on this work and published a number of papers were we have made it possible to make use of the replay debugging technology also for standard realtime operating systems and development environments [13].

There exist other approaches that do not rely on special compilers or hardware, but they can only replay concurrent program execution events like rendezvous (after code transformations) [3, 14]. This means that they cannot handle real-time specific events like preemptions, asynchronous interrupts or mutual exclusion operations.

4 Conclusions

In this article we have outlined a framework for increasing efficiency and quality in embedded real-time system testing. In addition, we have briefly described the contents of the framework (a number of analysis methods

suitable for implementation within the framework). As the next step in our research, we intend to formalize the analysis methods described in this paper (up to date, the algorithm for deriving the ICFG of a real-time system has been formulated and submitted for publication). Furthermore, we intend to implement the methods as prototype tools. Using these methods and tools, our intention is to perform two case studies. The first of these case studies will be performed in conjunction with an undergraduate course in real-time systems at Mälardalen University. This study will be aimed towards small, embedded real-time robotics systems. The second case study will be performed on a full-scale industrial real-time system. It is our intention that some of our methods at this point have evolved to such an extent that they are able to handle more complex systems (e.g., vehicular systems or robot controllers).

References

[1] T. Ball. On the Limit of Control Flow Analysis for Regression Test Selection. In *Proceedings of the ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 134–142, 1998.

[2] B. Buck and J. K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[3] R. H. Carver and K.-C. Tai. Replay and testing for concurrent programs. In *IEEE Software*, volume 8(2), pages 66–74, 1991.

[4] Z. Chen, B. Xu, H. Yang, K. Liu, and J. Zhang. An Approach to Analyzing Dependency of Concurrent Programs. In *APAQS '00: Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*, page 39, Washington, DC, USA, 2000. IEEE Computer Society.

[5] S. Chung, H. S. Kim, H. S. Bae, and D. G. L. Yong Rae Kwon. Testing of concurrent programs after specification changes. In *Proceedings IEEE International Conference on Software Maintanence (ICSM '99)*, pages 199–208, 1999.

[6] J. Gait. A probe effect in concurrent programs. In *Software - Practice and Experience*, volume 16(3), pages 225–233, Mars 1986.

[7] A. Möller, J. Fröberg, and M. Nolin. Industrial Requirements on Component Technologies for Embedded Systems. In *Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE7)*, pages 146–161. Springer Verlag, May 2004.

[8] NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing., May 2002.

[9] H. D. Pande and W. Landi. Interprocedural Def-Use associations in C programs. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 139–153, New York, NY, USA, 1991. ACM Press.

[10] A. Pettersson and H. Thane. Testing of Multi-Tasking Real-Time Systems with Critical Sections. In *Proceedings of Ninth International Conference on Real-Time and Embedded Computing Systems amd Applications*, Tainan City, Taiwan, R.O.C, 18-20 February 2003.

[11] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. In *Proceedings. Communications of the ACM*, volume 41(5), pages 81–86, 1998.

[12] W. Schütz. Fundamentals issues in testing distributed real-time systems. In *Real-Time Systems*, volume 7, pages 129–157, Boston, 1994. Kluwer Academic Publisher.

[13] D. Sundmark. Deterministic Replay Debugging of Embedded Real-Time Systems using standard components. Licentiate Thesis 24, Mälardalen Real-Time Research Centre, MRTC, March 2004.

[14] K.-C. Tai, R. Carver, and E. Obaid. Debugging Concurrent Ada Programs by Deterministic Execution. In *IEEE Transactions on Software Engineering*, volume 17(1), pages 45–63, January 1991.

[15] H. Thane and H. Hansson. Using Deterministic Replay for Debugging of Distributed Real-Time Systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, June 2000.

[16] H. Thane and H. Hansson. Testing Distributed Real-Time Systems. In *Journal of Microprocessors and Microsystems*, pages 463–478. Elsevier, 2001.

[17] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 86–96, New York, NY, USA, 2002. ACM Press.

[18] J. J. P. Tsai, K.-Y. Fang, and Y.-D. Bi. On real-time software testing and debugging. In *Proceedings of Fourteenth Annual International Computer Software and Application Conference*, pages 512–518, Oct 1990.

[19] F. Zhu, S. Rayadurgam, and W.-T. Tsai. Automating regression testing for real-time software in a distributed environment. In *Proceedings of First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 98)*, pages 373–382, 20-22 April 1998.