Automated Modeling of Real-Time Implementation

Peter Krogsgaard Jensen

BRICS Report Series RS-98-51

ISSN 0909-0878 December 1998

Automated Modeling of Real-Time Implementation

Peter Krogsgaard Jensen
BRICS[*]
Department of Computer Science
Aalborg University, Denmark
pkj@cs.auc.dk

December, 1998

Abstract

This paper describes ongoing work on the automatic construction of formal models from Real-Time implementations. The model construction is based on measurements of the timed behavior of the threads of an implementation, their causal interaction patterns and external visible events. A specification of the timed behavior is modeled in timed automata and checked against the generated model in order to validate their timed behavior.

1 Introduction

When developing a Real-Time application it is a problem to obtain precise information about how much CPU time is needed to complete the jobs of the application. A widely used way to make schedulability analysis is to use an offline worst case execution time (WCET) calculation. However, when several processes (or threads) interact via shared data, this calculation often becomes extremely complicated. This problem has been addressed in [9] using a framework where the offline analysis is extended with some application dependent knowledge and use of priority

[*]Basic Research in Computer Science,
Centre of the Danish National Research Foundation.

inheritants protocol. The work described in this paper is directed towards automatic collection of application dependent knowledge, resulting in less manual (and error prone) work to do schedulability analysis.

Besides schedulability, the logical correctness is also important to Real-Time applications. A number of formal tools are already available to support the correctness analysis during the design phase of such applications, but there is still a gap between design and implementation - and this may cause human errors. One approach is automatic code generation, but often the formal method is used only for essential algorithms and to model parallel composition. This makes it impossible to auto-generate the complete implementation. The work described here attempts to bridge the gap from implementation to design, by automatic synthesis of a formal model, which incorporates the actual (measured) timing behavior of the application. The model can then be fully analyzed by existing automated tools like e.g. UPPAAL [5].

In this work, we suggest a semi-automated iterative way to attack the above problems: First an initial implementation is developed and instrumented with the logging of relevant events; then a series of runs are logged and three different models are synthesized - including a timing diagram; finally the models are analyzed by using an automated real-time model checker and timing errors are corrected in the next iteration. The corrections may be validated by a new iteration.

It is our plan to implement tool support for the above method at a prototype level and to evaluate its feasibility through realistic case studies. In the present paper we present an preliminary result, i.e. we present our event logging tool and the tool for generating timing diagrams. Also, we sketch how to derive the models to be used by the model checker, and we present the preliminary experiences on a non-trivial case study. The prototype tool does not support testing, but assumes that the log stream it experiences is sufficient for creating a complete model.

A result obtained with the prototype tool, is the automatic calculation of the average case execution time (ACET). ACET is calculated as the average of a set of execution times, between f.ex. job start and job end. The ACET therefore becomes a number for how much CPU a particular job needs. Another result is the deducting of timed behavior which is pictured in a timing diagram, called execution time graph (ETG), this is done to present an overview of the interaction pattern between threads. The ETG is in fact an annotated message sequence diagram, where both synchronous and asynchronous interaction is pictured. This is the current state of the prototype tool.

The work done in by Havelund, Skou & Larsen in [3] indicates that it is possible to verify time requirements in a single processor interleaved system, and in this way an alternative to offline schedulability analysis is obtained.

The work described here is based on a series of tests of an industrial process control application, performed on a single CPU system using the RT-Mach micro kernel. The soft- and hardware system is described in section 2. In section 3 is a description of the necessary analysis to generate the implementation model, which still remains to be fully defined. Finally, in section 4 is located a plan for the future work.

2 The RT Test System

The target system is a single CPU running an RT-Mach micro kernel extended with event logging on both kernel and user level. The software system consist of four parts: event logging subsystem, submarine test application with testbed, ACET analysis prototype tool, and the UPPAAL model checker. The event logging subsystem, the prototype tool and the UPPAAL model checker are application independent, and will analyze any instrumented application running on RT-Mach. Figure 1 shows the data flow in the software system, but before going through this the main components are described individually.

The event logging subsystem is a part of the RT-Mach micro kernel, and can log scheduling- and user-events with a time stamp local to the machine. The event logging subsystem is developed by the RT-Mach group at CMU, and has been used by several tools. The system has been customized by the author to connect it to the prototype tool. The logging is fast and best effort, but congestion and packet loss is handled by dropping affected sub-traces during analysis.

The submarine test application is a 4000 lines multi threaded program set. It is a small process control system, where an unmanned submarine is directed from a ship. The submarine system handles a variety of periodic jobs automatically, and it receives sporadic commands from an operator at the command bridge. The application has been instrumented with a small number of system calls for logging. *A testbed* controls the input to the application under test, making it possible to simulate different types of situations and errors in the submarine environment. This work is trying to improve test analysis, by automating critical tasks, therefore we assume that the application is put through a sufficiently thorough test. The log information we analysis is then assumed to come from such test. In out example the testbed is used to drive the application through a realistic series of runs, such that all types of jobs are executed, and the different input is impressed on the application many times with random intervals. We will not further elaborate on what a sufficient test is, as this is not the scope of the paper.

The ACET analysis prototype tool is used to observe the system. Job knowledge, originating from the log information, is used to deduct job-patterns and to book the time spent to the correct job. How to produce a formal implementation model is described in section 3.

The UPPAAL *model checker* is an automatic model checker working on timed automata (TA). It incorporates Real-Time clocks as well as discrete analysis. This highly specialized tool is described in [5].

In figure 1 the dashed line divide application dependent and independent parts of the system. The dotted line is the network boundary, where the left part is executed on the target computer, the right part is spread on the adjacent network. All arrows are dataflow. The application exchanges directions, commands, information and alive signals with the testbed. The testbed is controlled by an operator, either interactively or it can be programmed to operate automatically. Via the system calls made by the test application, the kernel generates logging events and ship these of the local host. The stream of log events are received by the prototype tool which can store, calculate and display information about the logging events received. The tool can run in both automatic and manual mode. In manual mode a designer can take interactive control and generate ETG diagrams, and job- and thread-level models. The job- and thread-models are combined with the selected platform model and an operator defined requirement model in UPPAAL. The complete model can now be checked against its requirements by the analyst which is interactive with UPPAAL.

3 Building the model

The complete model consist of a requirement, an implementation and a platform-model. The requirement model contains external observable events and their timing constrains. An implementation model has two levels: job- and thread-level. The job level maintains information about the period or mean arrival time (MAT) for each job executed during test.

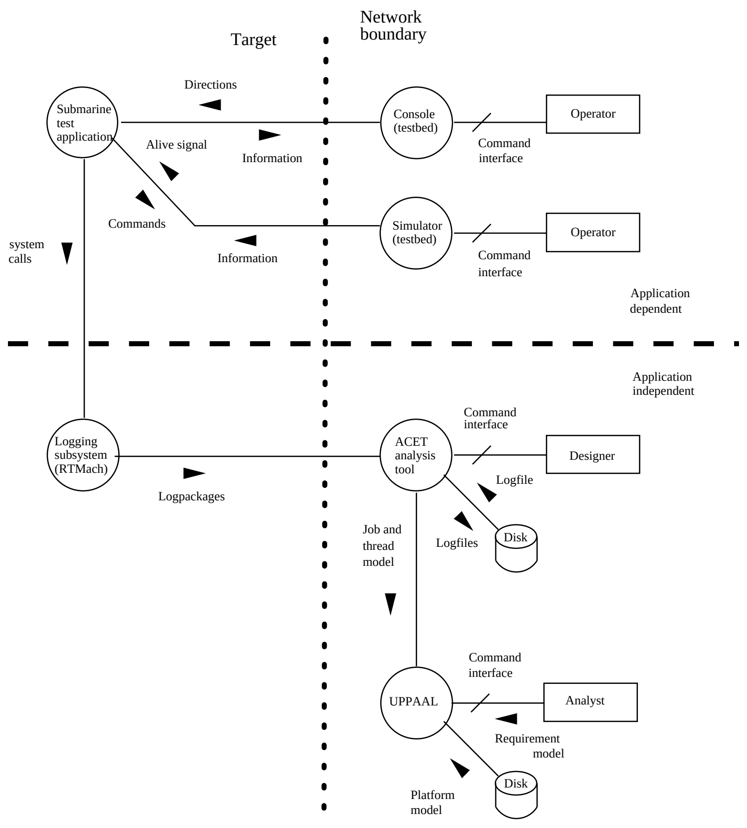

Figure 1: Dataflow in the experimental system.

The thread level describes the ACET and causal interaction patterns. The platform level contains the scheduling algorithm if needed.

The practical analysis performed in the prototype tool is divided in two layers, job analysis where job behavior is described in the job model, and ETG analysis where threads and interaction patterns are described in the thread model. The requirement- and platform-model are more static and will be created manually, once for each application/platform.

3.1 Application assumptions

In order to make the analysis we must assume that the application is a set of threads each responsible for one or a set of clearly defined task(s) - like “listen on network”, “transmit on network”, or “do calculation A”. We also assumes that the application will solve a job, by using the same threads in the same sequence for each repetition of the job. Furthermore we assume that a thread, which uses a resource, will use the same resource for each repetition of the job. These assumptions enables us to view the work done by a Real-Time application as a set of skeletons, and the analysis described here will synthesis these skeletons. Further it will calculate how often a skeleton is used, how much CPU it consumes, and what resources it accesses.

3.2 Job Model

To describe the job behavior of an arbitrary Real-Time application, a connection must be established between the threads of the implementation and the specification defining the job requirements. This is done by instrumentation, such that a thread, during execution, will state which job it is working on, and further log important (external observable) events. A job trace is created when events are assembled from all the threads participating in the job execution. For each job type the job model must know the frequency, and it is found by calculating the period, or MAT and standard deviation from time stamping of the job traces. This is enough information to produce a job model which will reflect the series of runs the application experienced.

3.3 Thread Model

To describe each thread of the application, its ACET and interaction with other threads must be modeled. The ACET is needed to model the CPU consumption, and the interaction patterns between threads are needed because they will restrict the computation. From a job trace a skeleton of the interaction can be extracted, be examining the use of mutexes and semaphores, the message passing, and the IPC. All job traces with the same skeleton are concentrated into one ETG, using the ACET - in place of the WCET - as the measure for how much CPU a certain job needs. It is now possible to create an automaton for each thread (in the ETG), and the set of automata will describe the interaction of the threads when they are working for a certain job.

When this is done for all jobs in the application, the behavior of each thread is completely described, and the thread model will constrain the model checking such that only the implemented behavior is possible.

3.4 Model checking

To complete the description of our Real-Time system, a platform model is needed. It will be application independent, but must incorporate the scheduling algorithm. With this method it is possible to use different scheduling algorithms and even verify the implementation model on a non-existing platform.

The model checking is done on a requirement model, consisting of a set of timed automata which define the end-to-end time requirement with respect to the external observable events. Figure 2 shows an example model, where a sporadic event must be answered within 1.0 second. A question to the model is whether it is possible that Msg-Out is not done before t equals 1.0 - or even worse is it possible that Msg-In can happen without Msg-Out happens afterwards.

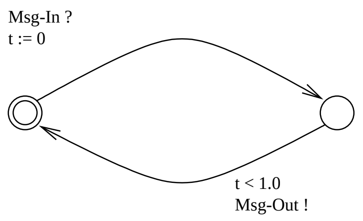

Figure 2: Requirement model expressed as a timed automaton for a time requirement where a sporadic event Msg-In must be answered with Msg-Out within 1.0 second. The implementation model is responsible for generating the matching events as the model is synchronous.

During model checking the job model is responsible for initiating jobs, the thread model restrict sequences of interaction, the platform model restrict CPU usage, and the requirement model defined the questions that must be examined. Finally it is left to the model checker to go through all allowed computations, and possible finding erroneous, or perhaps more efficient computations, that those actually seen during test.

4 Future work

Work is currently done, to automate the generation of the implementation model. The logging and analysis of traces is completed, while the interaction patterns remains to be incorporated. The logging subsystem must reveal detailed information about mutex access and the type of thread-to-thread call. In particular the thread-to-thread call is interesting because several different types of synchronous and asynchronous call/messages are possible. A plausible solution is to create a piece of middleware through which the applications must call to interact with each other. This enables an application independent logging.

Having seen that it is feasible to log information from a running Real-Time application, we must address the question of how our observation changes the original system. It is changed in two ways: extra code complexity during development of the application, and extra CPU cycles during execution. The overhead added to the design phase is small calculated as extra lines of code. The CPU overhead still remains to be measured, as we are still making changes to the RT-Mach kernel.

5 Acknowledgement

The described work is going on at Aalborg University under supervision of Professor Arne Skou.

References

[1] C. M. Chen Lee, Katsuhi Yosida and R. Rajkumar. Predictable communication protocol processing in real-time mach. *Proceedings of Real-Time Application Symposium*, 1996.

[2] D. Haban and K. G. Shin. Application of real-time monitoring to scheduling tasks with random execution times. *IEEE Transaction on Software Engineering*, 16(12), December 1990.

[3] K. Havelund, A. Skou, and K. G. Larsen. Formal verification of an audio/video power controller using the real-time model checker UPPAAL. *Work in progress*, 1998.

[4] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the timing analysis of pipelining and instruction caching. *Proceedings of Real-Time Systems Symposium*, December 1995.

[5] K. G. Larsen, J. Bengtsson, F. Larsson, P. Pettersson, and W. Yi. UPPAAL - a tool suite for automatic verification of real-time systems. *Proceedings of the 4th DIMACS Workshop on Verification and Control of Hybrid Systems*, October 1995.

[6] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, May 1994.

[7] J. E. Sasinowski and J. K. Strosnider. Artifact: A platform for evaluating real-time window system designs. *Proceedings of Real-Time Systems Symposium*, December 1995.

[8] L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(7), 1990.

[9] L. Sha, R. Rajkumar, and S. S. Sathaye. Generalized rate-monotonic scheduling theory: A framework for developing real-time systems. *Proceedings of the IEEE*, 82(1), January 1994.

[10] H. Tokuda and P. Rao. Real-time mach: Towards a predictable real-time system. *Proceedings of the USENIX Mach Workshop. Burlington, Vermont: The USENIX Association*, 1990.

[11] A. Wellings and A. Burns. *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems.* ELSEVIER, Amsterdam, Netherlands., 1995.

Recent BRICS Report Series Publications