4th International Conference on Leveraging Applications of Formal Methods, Verification and Validation, Heraclion, Crete, 2010, Special Track on Resource and Timing Analysis

Towards an Evaluation Infrastructure for Automotive Multicore Real-Time Operating Systems

Jörn Schneider, Christian Eltges

Dept. of Computer Science
Trier University of Applied Sciences
Trier, Germany
{j.schneider, c.eltges}@fh-trier.de

Abstract. The automotive industry is on the road to multicore and already included supporting features in their AUTOSAR standard, yet they could not decide for a multicore resource locking protocol. It is crucial for the future acceptance and possibilities of multicore systems to allow for informed decisions on this topic, as it immediately impacts the inter-core communication performance and thereby the value-cost ratio of such systems. We present the design of a real-time operating system simulator that allows to evaluate the different multicore synchronisation mechanisms of the real-time research community regarding their fitness for automotive hard real-time applications. You can reuse the key design idea of this simulator for any simulation based tool for the early timing evaluation of different real-time mechanisms, e. g. scheduling algorithms.

1 Problem

A challenging application area for multicore systems are hard real-time systems in cars, e. g. electronic control units for airbags, electronic stability control, and driver assistance systems. The automotive industry recently released the specification of a first multiprocessor real-time operating system (RTOS) as part of the AUTOSAR standard [1]. The concept specifies a partitioned system with tasks and interrupt service routines statically mapped to cores. Each core runs a fixed priority scheduler for its particular task set and tasks can be activated across cores. Transferring data is based on sharing memory between cores.

But how can this be done without deadlocks, priority inversion and unbounded remote blocking? Traditionally, resource locking protocols are used in real-time systems to achieve this. In the automotive domain the immediate ceiling priority protocol (referred to as OSEK Ceiling Priority Protocol) is readily available in any OSEK or AUTOSAR compliant RTOS. Yet, this works for uniprocessor systems only.

The responsible AUTOSAR subcommittee, which was then headed by the first author of this paper, initially planned to introduce a multicore resource locking protocol for this release of the operating system specification. However, it turned out that this goal was too ambitious given the short deadline

for AUTOSAR release 4.0. The requirements document [2] still reflects the ambitious goal, yet the specification itself clearly changed in this regard after the responsible person for the topic changed. AUTOSAR release 4.0 lacks a multiprocessor synchronization mechanism that is suitable for hard real-time systems — only spin locks are supported.

One reason for this shortcoming is that it is unclear which resource locking approach is most suited to fulfil the requirements of the industrial practice. Clearly, the performance impact plays a central role here. Investigating this impact requires to look at two aspects. First, the blocking behaviour and second the implementation overhead. The former one depends on the resource locking characteristics of the application in combination with the particular resource locking protocol. The quantity of the latter additionally depends on the chosen realisation of the resource locking protocol within the RTOS. Naturally, the timing aspects of the chosen protocol might seriously impact the communication efficiency. Because the main value of multicore systems for the automotive domain lies in its better cost-performance ratio, the timing behavior of the cross-core communication is crucial for the adoption of the new hardware concepts.

The automotive industry cannot afford to go through the painful process of implementing different resource locking protocols, applying them in various products, and making enough experience to differentiate good from bad approaches. We designed a dedicated simulator to address this problem.

2 The Proposed Simulator

The key design idea that distinguishes our simulator from similar approaches is to consequently separate the two concerns *simulated functionality* and *simulated time*. In other words, if you specify a new mechanism to be simulated, e. g. a resource locking concept, you have to specify the algorithm and its temporal behaviour separately. You might think that this is an unnecessary and cumbersome complication. But it even reduces your workload, if you use the simulator for the intended purpose.

When considering the basic functionality of resource locking protocols (or real-time operating systems as a whole) it becomes quite evident that there are a lot of fundamental operations that need to be performed regardless of the specific version. For instance, when deciding which task is to run next, it is in most cases necessary to pick the task with the highest priority. Various implementations of the operation *identify highest priority task* are possible, yet the result is always the same. However, the timing might be completely different. When you use our simulator you can simply take a provided library function *getHighestPriorityJob* and specify the timing behaviour of the particular implementation you have in mind without even implementing it (see Listing 1.3 and 1.4 below for examples). To compare two RTOS A and B with compatible APIs but different implementations you just bind the corresponding timing functions to the API functions for each simulation run.

We believe that the idea to separate *simulated functionality* and *simulated timing* can be widely reused in simulation based tools for the early timing evaluation of different real-time mechanisms, e. g. scheduling algorithms. As we conjecture this makes such tools much more usable for different research groups to compare their approaches on equal terms.

Our simulator allows to define tasks in an abstract language (instead of C code). The only things to be specified in this language for a given task are the scheduling relevant calls to API-functions and sections of the task that, from a scheduling point of view, just consume time. An example that shows this idea can be seen in Listing 1.1. It defines a task that executes some calculations that do not influence the scheduling (denoted by the *time x;* expressions) and locks a resource RES_1 for 5000 processor cycles.

Listing 1.1. Definition of a task

```
Task t1 = do {
    time 10000;
    GetResource RES_1;
    time 5000;
    ReleaseResource RES_1;
    time 3000;
    TerminateTask;
}
```

Internals of API-calls can be specified with the help of *basic functions*. These are the minimal building blocks of the simulator language. Listing 1.2 gives an example API-call implementation using basic functions, like *getHighestPriorityJob* and *setState*.

Listing 1.2. Definition of an API-call

```
TerminateTask = do {
    setState currentJob SUSPENDED;
    j <- getHighestPriorityJob;
    dispatch j;
}
```

To compare different protocols, the execution times of the basic functions are specified via timing functions for each protocol. When the simulator executes a basic function, it calculates the number of cycles via the corresponding timing function.

This makes it possible to have different timing functions that simulate different implementations, without reimplementing the basic functions. For example, consider the *getHighestPriorityJob* function. The priority queue could be implemented as an unsorted list or as a sorted list. In the first case, getting the highest priority job would take O(n) steps, in the second case it would take O(1) steps. Examples are given in Listing 1.3 and Listing 1.4.

Listing 1.3. Timing function for linear runtime

```
linearTimeOfGetHighestPriorityJob
    return 10 + 5 * length(readyQueue)
```

Listing 1.4. Timing function for constant runtime

```
constTimeOfGetHighestPriorityJob
    return 10
```

Note that timing functions can use the complete state of the simulated code to derive the proper execution time for each calling context. This feature is used in Listing 1.3 to consider the current length of the ready queue.

3 Related Work

A multitude of simulators for different purposes were implemented by research groups or companies. Naturally we did not investigate all of them and the ones we investigated were usually different in many important aspects, like timing granularity and so on. None of them seems to follow our concept of consequently separating the two concerns *simulated timing* and *simulated functionality*. At least one simulator nevertheless should be mentioned here. RTSSim [3] is closely comparable with our work, it has an even broader scope regarding its intended usage. The key differences are that we use an abstract language instead of C, that we support multicore, and that our approach consequently separates the two concerns *timing* and *functionality*.

4 Conclusion

We presented the design of a novel simulator for the specific purpose of evaluating multicore resource locking protocols for their fitness to be used by industry in automotive electronic control units. The key idea to separate functional realisation from simulation time is one that, as we believe, is well suited to be used in many simulation approaches in the field of timing analysis. We hope that the final simulator as well as the simple idea of separating function and timing will be (re-)used in the research community.

References

1. AUTOSAR release 4.0 — Specification of Multi-Core OS Architecture (12 2009), http://www.autosar.org/download/R4.0/AUTOSAR_SWS_MultiCoreOS.pdf
2. AUTOSAR release 4.0 — Requirements on Multi-Core OS Architecture (11 2009), http://www.autosar.org/download/R4.0/AUTOSAR_SRS_MultiCoreOS.pdf
3. Kraft, J.: RTSSim - a simulation framework for complex embedded systems. Technical Report, Mälardalen University (March 2009), http://www.mrtc.mdh.se/publications/1629.pdf