SimTrOS: A Heterogenous Abstraction Level Simulator for Multicore Synchronization in Real-Time Systems

Jörn Schneider, Michael Bohn, Christian Eltges
Dept. of Computer Science
Trier University of Applied Sciences
Trier, Germany
{j.schneider, m.bohn, c.eltges}@fh-trier.de

Abstract—To provide a common ground for the comparison of real-time multicore synchronization protocols we developed a framework that supports heterogenous levels of abstraction for simulated functionality and simulated timing. Our intention is to make the simulator available to the real-time research community and industrial users. For the latter we initially focus on automotive real-time systems. This paper describes the simulation framework and the novel idea of heterogenous abstraction levels that lies at the heart of its design. Notwithstanding the clear focus, we believe that the simulator itself as well as the concept of heterogenous abstraction levels can be useful in a significantly broader way.

Keywords—real-time, operating system, simulator, multicore, abstraction, synchronization

I. INTRODUCTION

For real-time researchers it is important to demonstrate the benefits of novel approaches in comparison to previous ones. However, it can be a tedious job to achieve a comparison on equal terms, since this usually requires implementing concepts and evaluation infrastructures of other groups. Moreover, the transferability of the evaluation results to an industrial context is quite limited, as the original comparison ignores the caveats of a particular product context for good reasons. Unfortunately, important qualities (e.g. memory consumption, temporal implementation overhead, or energy demand) of a solution are often sensitive to the ignored factors, thus rendering too general results useless.

We believe that many valuable concepts of our community are due to the lack of scalable and transferable evaluations not appreciated in the way they would deserve and the adoption by industry is significantly smaller and slower than it could be. As it is infeasible to investigate each solution a priori in every potential industrial usage scenario, the comparison approaches should allow for a fast re-exploration considering a concrete product context.

We developed the simulation framework described in this paper to tackle these issues for multiprocessor resource locking protocols, especially in the context of automotive real-time systems. Notwithstanding this clear focus, we believe that the simulator core itself can be used for any timing evaluation of multicore real-time systems and moreover, that the novel idea of heterogenous abstraction levels that lies at the heart of its design can also be a key to fast re-exploration when investigating further runtime properties such as memory or energy consumption.

Efforts to migrate legacy applications to multicore systems, such as the one sketched in [1], which led to the development of the described simulator, require sound multicore synchronization mechanisms. Moreover, automotive industry identified in its AUTOSAR real-time operating system standard the need for resource locking protocols in future multicore systems [2]. However, none of the available approaches (e.g. [3], [4], [5]) has been chosen so far, nor was any suitable replacement incorporated into the standard.

Comparison studies such as the ones by Brandenburg, and Anderson [6] and by Lakshmanan, Niz, and Rajkumar [7] are of no help for the automotive industry as they come to diverging conclusions. One result is in favor of spin-based the other in favor of suspension based schemes. In our opinion it is evident that the question which multicore resource locking protocol is better can only be decided for a given characteristic of an application and a given implementation of the protocol in a particular operating system, e.g. AUTOSAR. Furthermore, we conjecture that the HW characteristics of automotive systems such as low processing power can have a significant effect on the outcome of this question.

As briefly described in [8] our SimTrOS simulator is designed to deliver detailed results about the suitability of different protocols and their implementation overhead for the specific characteristics of automotive applications. The design of the simulator was driven by the following central requirements:

1) Modelling of applications and operating system services, e.g. multicore resource locking protocols, shall be decoupled
2) The two simulation concerns functionality and temporal behavior shall be strictly separated
3) Any simulation result shall be deterministically reproducible, i.e. even if a system with race conditions is

http://www.fh-trier.de/go/prosymig
simulated, the simulator shall produce the same output if it receives identical input. Otherwise incremental changes of the simulated system might become indistinguishable from random effects. Note that variability of the simulation can be achieved by using different seed values for pseudo random number generation.

II. RELATED WORK

Since the core of the simulator is a discrete event simulation engine, it shares the basic characteristics of these types of simulators. We investigated several of them and found two crucial aspects that none of them addressed in the required combination:

Temporal Granularity: Essential timing issues determining the efficiency of multicore synchronization lie at the RTOS implementation level and even at HW level. A suitable simulator has to support the investigation of these properties, and many simulators do this.

Abstraction Level: The intended users of the simulator shall model the functionality of applications or resource locking protocols at convenient levels of abstraction.

The problem is that simulators that allow for fine grained timing investigation require the user to specify all the functional details at the corresponding low level of abstraction. To address this issue we developed the idea of a simulation framework with heterogenous abstraction levels as it is explained in Section III.

Although we found no simulator that is comparable in this respect, two simulators should be mentioned here. RTSSim [9] is a close match regarding other aspects. It uses a system model written in C-Code, where the actual simulation is running the compiled code, similar to our approach. The key differences are that we use an abstract language instead of C, support multicore, and consequently separate the two concerns simulated timing and functionality. The other simulator RTSim [10] is also similar and accepts definitions written as C-Code. The main difference to our solution is again, that our simulator differentiates strictly between timing and functionality.

III. NOVELTY OF THE SIMULATOR

The main contribution of this paper is describing a simulation framework that consequently separates different levels of abstraction. It is designed to compare the performance of multicore resource locking protocols from a research or an industrial perspective. Researchers can utilize the simulator to quickly investigate crucial properties of their multicore synchronization approaches such as the question under which basic assumptions is one concept better than another. Practitioners can benchmark different multicore resource locking protocols against one another for a particular application and operating system context in a rapid prototyping fashion. The simulator is implemented in Haskell but requires no user knowledge about functional programming.

A major advantage of the simulator is that it provides a fast and easy way to investigate the same protocol with different underlying timing models andvice versa. This is achieved by the simulators ability to strictly separate the two concerns simulated functionality and simulated timing. In other words, the simulator works with heterogenous abstraction levels for model properties. The timing property can be changed without altering the functional model and the specification of the protocol algorithms can be modified while keeping the same basic timing model.

The benefit of separating the concerns functionality and timing can be illustrated by the following example. The performance of the ready queue management of the scheduler could make a significant difference as it contributes to the overhead of suspension-based schemes. In a normal simulation environment investigating the difference between algorithms maintaining a sorted or unsorted list of jobs would require to implement a complete RTOS scheduler in two versions. Our simulator allows to consider this by just specifying a different temporal behavior while the scheduler is described at a high level of abstraction with the same standard queuing algorithm in any case. Consider how much easier it is for an user of our simulator to evaluate different protocols and implementation flavors of the same protocol in comparison to the state-of-the-art simulation environments.

IV. SYSTEM MODEL

The simulator distinguishes between events and effects of events. An event is a point in time and an effect is a state change. Note that each event has one assigned effect, that an effect can leave the current state unchanged, and that more than one event can happen at the same time.

Periodic and one time events are supported. Periodic events have two parameters: period (reoccurrence interval of events), and phase (time offset for the event sequence). One time events are periodic events with period $\infty$.

Events are assigned to locations according to their origin. The current framework implements cores and the environment (anything outside of cores) as separate locations. Events from the environment are referred to as external events throughout the paper.

On top of the basic concepts of the simulator itself (i.e. events, effects, and locations) an arbitrary model of the simulated system can be established. So far we implemented an AUTOSAR compliant system model with its inherent restrictions. Note that these restrictions, such as a partitioned fixed priority scheduler, are by no means limitations of the simulator itself.

http://www.fh-trier.de/go/prosymig
V. ARCHITECTURE AND IMPLEMENTATION

A. Architecture

The architecture of the simulation framework is depicted in figure 1. The top layer consists of tasks, interrupt service routines (ISRs), and external events. We denote this set of entities the application set.

The application layer uses functionality from the underlying API-call layer. The API-call layer provides two sets of functionality. The first set consists of operating system functionality of the simulated system, classically denoted as system calls. The second set of functionality are helper functions that are provided by the simulator.

As applications are composed of API-calls, API-calls are again composed of more primitive functions, the basic functions. Basic functions interface directly with the simulator core and only they can directly manipulate the system state of the simulator. Thus the basic functions are the foundation of the simulated functionality. The temporal behavior is specified separately by timing functions. By combining a basic function and a timing function the functional and temporal dimension of executing the simulated code can be freely defined.

The lowest layer consists of the simulator core that is responsible for the control of the simulation.

The first two layers are separated to ease the use by two developer groups: Users specify applications on top of predefined API-calls. Implementors create abstract implementations of real-time operating systems as API-calls for users to operate with. The distinction between these two groups is of course purely virtual, as one and the same person could be interested in both aspects.

Although the simulator is implemented in Haskell and running a simulation actually means executing compiled Haskell code, users and implementors use only a small subset of the Haskell language. The syntax of this abstract programming language is quite simple and requires no knowledge of the Haskell language.

B. Implementation of the Simulator Core

The simulator core is a discrete event simulation engine that selects one event after another and skips time periods for which no events occur. As the origin of an event is either a processor core or the environment, there are two cases where the system state of the simulator can be changed: finishing the execution of a basic function or the occurrence of an external event.

The simulator computes at which point in time the next event occurs by considering all modeled locations that might produce events. External events are explicitly defined, e.g. every 10 time units event x occurs. The finishing time of a basic function is calculated via the attached timing function. If the nearest future event is the only one that happens at this point in time the simulator selects it as next event to be processed. We first describe an example for these simple cases before considering the situation of two or more events at the same time.

1) Single Events: Consider the single core scenario shown in figure 2. We use the following notation

$$\sigma : \text{SystemState}$$

is the simulator internally used state of the system,

$$t : \text{Time}$$

is the simulated time,

$$bf_1, bf_2 : \text{SystemState} \rightarrow \text{SystemState}$$

are basic function$_1$ and basic function$_2$, and

$$tf_1, tf_2 : \text{SystemState} \rightarrow \text{Time}$$

are used to refer to the timing function attached to $bf_1$ and $bf_2$, respectively. Note that although it is possible to combine different timing functions with the same basic function at different calling points we refrain from using this ability in the examples of this paper.

The simulation starts with time $t = 0$. The next point in time where something happens is $t = 1$ where an external event occurs. The simulator advances $t$ to 1 and derives the effect of the external event (by computing the attached
function of type $\text{SystemState} \rightarrow \text{SystemState}$). In this example, the event causes $J_1$ to be the new running job. $J_1$ executes $bf_1$.

To determine the next event the simulator calculates the nearest point in time when an event will happen. At time 3 an external event occurs and $bf_1$ would finish execution at time $t + tf_1(\sigma) = t + 4 = 5$. Since the external event occurs first, the simulator picks it and computes its effect after setting $t = 3$. As the event represents an interrupt request, the computed effect is starting an interrupt service routine (ISR). The ISR (consisting of $bf_2$) preempts $bf_1$. The simulator calculates $tf_2$ for the current system state and derives 2 time units as the execution time of $bf_2$. Thus the next time point for an event is $\min(5, \infty)$ as no further external events are specified.

$t$ is advanced to 5 where $bf_2$ finishes execution and the simulator computes the effect of this basic function on the system state, i.e. $bf_2(\sigma)$. The only effect of $bf_2$ is to return from the ISR, i.e. restoring the previously running job. $bf_1$ therefore continues execution. As this basic function already executed 2 time units, the simulator calculates the finishing time to $t + (tf_1(\sigma) - 2) = t + (4 - 2) = 7$. Note that $tf_1$ is applied to the current system state and thus could deliver a different execution time than beforehand, which is not the case in this example.

Thus the next (and last) event that occurs in the simulation happens at time 7 when $bf_1$ finishes execution. The effect of this basic function is applied to the system state. The simulator stops, since the next point in time where something happens is $\infty$.

2) Observations: As demonstrated in the example, effects of external events and basic functions are always applied atomically. This prevents race conditions in the simulator data structures and simplifies the implementation of basic functions. Furthermore, effects of events take zero time.

With the strict separation of basic functions and timing functions, different basic function implementations can be simulated without actually touching the implementation of effects and by only changing the timing function. It is also possible to simulate theoretical cases, where a basic function would take no time at all, e.g. scheduling without scheduler overhead.

Because the absolute time a basic function requires is newly computed whenever the simulator calculates the next event, it is possible that the time already executed on behalf of this basic function, is greater than the newly computed time. In this case the simulator finishes the basic function at the current point in time. Consider for example a basic function $\text{getSpinlock}$, which takes no time if the lock is free, but infinite time if the lock is blocked. If a job tries to acquire the lock, but the lock is not free, the job still accumulates calculation time. When the lock is released, the time needed to get the lock becomes 0 and the lock is granted immediately, but is not retroactively granted in the past.

3) Multiple Events at the same Time: Even in the special case where only one core is simulated, two events can happen at the same point in time. In order to fulfill requirement 3 (deterministic execution) the order in which the events are applied has to be the same for every execution of the simulation. Therefore external events occurring at the same time as the completion of a basic function are always considered to happen before the basic function finishes, and all external events have a strict order.

Now consider the case of a multicore system where several basic functions could finish simultaneously on different cores. To solve this problem, the implementor has to define a function that decides which effect of a basic function should be applied first. A simple decision function could apply the basic function from the core with the lowest ID first.

The decision function can inspect all possible next system states and return the new global system state. Setting a system state as the new global system state triggers a new calculation of the nearest effect in the simulator. In case there is more than one effect remaining for this point of time, the decision function is consulted again. Regardless of the decision taken, a log message is written, indicating that a potential race condition of the simulated system was encountered.

Simulating a multicore system is not really different from simulating a single core system. In the single core case the simulator has to calculate the next point in time from two locations, the environment and the one core. In the multicore case the simulator has to consider $n + 1$ locations, where $n$ is the number of cores that are simulated. Note that it is possible that more than one basic function finishes execution at the same time on one core, e.g. if two basic functions execute in zero time. The simulator resolves this automatically by the position of the basic function in the task program, i.e. a basic function that was called before another basic function on the same core always applies its effect first.

VI. DEFINING A SIMULATION

In the following subsections we describe in more detail, how constructs from the three topmost layers are defined.

A. Defining applications

An example for a task definition in the simulator is depicted in listing 1. The most interesting property here is taskProgram. A task program defines the behavior of a task and is defined in the imperative language style mentioned before. The program here is strictly sequential, but the abstract task language also supports conditions and loops. A task program is mainly a series of calls to functions provided by the API-call layer.

The semantics of the abstract task programs depends on the API-call implementation. As in a concrete RTOS, the

http://www.fh-trier.de/go/prosymig
application developer only sees the interface to the operating system and depends on a detailed description of its exact behavior. The example uses an AUTOSAR based semantics.

A useful function to abstract arbitrary computations is the API-call time. It consumes time without changing the state of the simulated system.

Listing 1. Task Example

```haskell
task_i = autosarTask { 
    taskPeriod = 100, 
    taskPhase = 0, 
    taskPriority = 1, 
    taskName = "task_i", 
    taskCore = 0, 
    taskProgram = do { 
        osGetResource "R1"; 
        time 33; 
        osReleaseResource "R1"; 
        time 5; 
        osTerminateTask; 
    } 
}
```

The task description template is a wrapper around the primitives event and effect. The task definition in the example is a shortcut for the definition of an event with period 100 and phase 0. The effect of the task definition is then to start an ISR that activates the task on processor core 0.

The definition of an event is very similar to the definition of a task. An example for an event definition is depicted in Listing 2 (in fact this example is an alternative way to activate a task). Like a task, an event has a period, phase and a name (but no priority and core ID). In the example, a unique event that occurs at time 70 is defined. The main distinction between a task and an event is the property eventEffect. Where a taskProgram consists of a series of statements, an eventEffect is more general and can be any Haskell function from SystemState → SystemState.

Since we do not require Haskell knowledge from users, we provide primitive functions that should cover the most basic cases of system state manipulations due to events. In the example an ISR on core 1 is started with the helper function startISR. The second parameter can be an arbitrary, abstract program that defines the behavior of the ISR. Another typical case for the effect of an event is changing a variable in the system state. The helper function updateGlobalVar can be used to change the value of a global variable.

Listing 2. Event Example

```haskell
event_j = event { 
    eventPeriod = Infinity, 
    eventPhase = 70, 
    eventName = "event_j", 
    eventEffect = startISR 1 ( do { 
        osActivateTask task3 
    } ) — interrupt on core 1 
}
```

B. Defining API-calls

API-calls are defined in a similar syntax as task programs. Listing 3 shows an example for the definition of osTerminateTask, which was used in listing 1. API-calls are composed of basic functions or other API-calls. Note that the function schedule is not a basic function, but a composite API-call. Composite API-calls are defined like normal API-calls. What distinguishes them from normal API-calls is that they are only used internally and are not intended to be used in task programs. To distinguish externally visible API-calls, i.e. system calls, from internal API-calls, we prefix the former with os. This is purely a convention and not mandatory.

Listing 3. API-call Example

```haskell
osTerminateTask = do { 
    setJobVar "state" Suspended; — suspend task 
    schedule; — reschedule
}
```

```haskell
schedule = do { 
    j <- getHighestPriorityJob; 
    setRunningJob j; — execute highest priority task
}
```

C. Defining basic functions

The layer below API-calls consists of two parts: functional behavior (basic functions) and temporal behavior (timing functions). The Basic function getHighestPriorityJob, used in Listing 3, searches the ready queue for the job with the highest priority and returns the corresponding job number as a result.

For the function getHighestPriorityJob one can imagine two implementation strategies: searching through an unsorted list or maintaining a priority ordered list of jobs. In the first case getting the highest priority job would take O(n) steps and in the second case it would take O(1) steps. We can capture these two different timing behaviors by defining the timing functions depicted in Listing 4.

Listing 4. Timing Function Example

```haskell
linearTime s = 10 + 5 * length (getL readyQueue s) 
constTime s = 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 4 to consider the current length of the ready queue.

To combine a concrete pair of basic and timing functions the primitive makeBasicFunction, as shown in listing 5 is provided.

Listing 5. Basic Function Definition

```haskell
getHighestPriorityJob = makeBasicFunction linearTime 
getHighestPriorityJobImpl
```

1Choosing one or the other implementation has of course an impact on the timing of the corresponding function addJobToReadyQueue. Adding a job to an unsorted list takes O(1) steps, adding a job to a sorted list takes at least O(log(n)) steps. Evaluating the impact of these kinds of decisions was the motivation for the development of the simulator.
D. Simulating a system

As described above, a system definition consists of various parts, which are possibly defined by different developers. Such a system, including application code, is compiled to a native executable in order to obtain a most efficient simulator. Since all definitions are pure Haskell code, we use the Glasgow Haskell Compiler (GHC) to create simulation executables.

The compiled executable accepts different options to control the simulation. It is possible to run a simulation interactively, simulating one step after another. In non-interactive mode a simulation is executed until completion or up to a specified time limit. During the simulation each call to a basic function and each occurrence of an external event is written to a log file.

VII. Conclusion

We presented the design of a novel simulator. The key idea of heterogenous abstraction levels allows to simulate a system with different timing parameters, without changing the functional behavior. This scheme was chosen to allow for a fast exploration of multicore real-time synchronization protocols in different implementation flavors for their employability in industrial automotive systems. We conjecture that the approach is equally suited to compare other mechanisms such as scheduling regarding their sensitivity to implementation overhead, e.g. context switches. Moreover, the idea of strict separation of timing and functionality is, as we believe, well suited to be used in many simulation approaches in the field of analysis of non-functional properties apart from timing.

The modular design of the simulator and its separation in application, i.e. task sets and external events, and operating system layer, i.e. API-calls, is another key feature aimed at easy usage. We intend to provide the simulator as an open tool to the research community and to successively build up a repository of simulation code for real-time operating systems and applications.

We started implementing different multicore resource locking protocols on top of an AUTOSAR system model. The next step is providing better ways to examine simulation results such as a tool that extracts statistics from the produced XML output of the simulator, e.g. average or observed worst-case execution, or blocking times of jobs. For visualization of simulator runs we plan to leverage existing tools, e.g. translating the output of the simulator to the input format of the Grasp tool [11]. A more remote goal could be to generalize the principle of heterogenous abstraction levels for the simultaneous investigation of multiple non-functional run-time properties such as energy demand, or memory consumption in addition to timing.

REFERENCES


