Time and memory tradeoffs in the implementation of AUTOSAR components

Alberto Ferrari
Parades
Via San Pantaleo 66
Roma, Italy

Marco Di Natale
ReTiS Lab.
Scuola Superiore S. Anna
Via Moruzzi 1, Pisa, Italy

Giacomo Gentile
and Giovanni Reggiani
Magneti Marelli S.p.A.
Via Timavo 33, Bologna, Italy

Paolo Gai
Evidence Srl
Via Carducci 64
Ghezzano (Pi), Italy

***Abstract*—The adoption of AUTOSAR in the development of automotive electronics can increase the portability and reuse of functional components. Inside each component, the behavior is represented by a set of runnables, defining reactions executed in response to an event or periodic computations. The implementation of AUTOSAR runnables in a concurrent program executing as a set of tasks reveals several issues and trade-offs because of the need to protect communication and state variables and to ensure time determinism. We discuss some of these tradeoffs and options and outline a problem formulation that can be used to compute the solution with minimum memory requirements executing within the deadlines.**

I. Introduction

The AUTOSAR development partnership, which includes several OEM manufacturers, Tier 1 suppliers, and tool and software vendors, has been created to develop an open industry standard for automotive software architectures. To achieve the technical goals of modularity, scalability, transferability, and function reusability, AUTOSAR provides a common software infrastructure. The current version of the standard includes a reference architecture and specifications for the definition of components and their interface.

The AUTOSAR project has focused on the concepts of location independence, interface standardization, and code portability. Although these goals are important, their achievement will not be sufficient. As with most other embedded systems, car electronics are characterized by functional and nonfunctional properties, assumptions, and constraints.

The current specification has at least two major shortcomings. The AUTOSAR metamodel, as of now, lacks a clear and unambiguous communication and synchronization semantics and a timing model. Similar to UML, the AUTOSAR metamodel is sufficiently mature in its static or structural part, but offers an incomplete behavioral description. Developers plan to remedy this with significant updates in the upcoming AUTOSAR revision, however.

Further, none of the standard's several layers address issues related to timing, performance and reliability. Applications' component interactions generate a variety of timing dependencies due to scheduling, communication, synchronization, arbitration, blocking, and buffering. Ensuring component reuse is not simply a matter of compile-time guarantees that the provided and required functions be accessible regardless of the software module's location and that formal parameters and actual parameters match. It also demands that the behavior of the reused components can be predicted in the new configuration and the result of the composition can be analyzed for timing faults. If developers fail to address this problem, the composition will eventually lead to possibly transient timing problems, including missed deadlines, task and message skipping, or overwriting and buffer overflows.

The definition of a timing model for AUTOSAR, and the development of a standardized infrastructure for the handling of time specifications, is the objective of the European Union ITEA 2 (Information Technology for European Advancement) project TIMMO (timing model). Started in April 2007, the project includes car manufacturers like Audi, PSA, Volvo Technology, and Volkswagen. The project targets networked automotive real-time systems with the goal of providing - a description language for time aspects in the development of automobile control units and networks, - a methodology for cross-company usage of this description language, and - a validation of the language by means of prototypical demonstrators.

Magneti Marelli, a car electronics company from Bologna, Italy is currently developing several projects for porting (some of) its applications to AUTOSAR[5]. The target application that triggered the considerations expressed in this paper is an automatic transmission control. The application requires flow preservation on the communication and is very sensitive to changes in the state variables and time jitter on the communication side. The application is running on a single processor and is characterized by timing constraints. The target platform also imposes tight constraints on memory resources (a serious concern during the development).

II. SW Components, runnables and tasks

In AUTOSAR, the *functional architecture* of the system is a collection of *SW Components* cooperating through their interfaces (Figure 1). Components interfaces are defined as

This research was supported in part by the European Commission under the projects INTERESTED: INTERoperable Embedded Systems Toolchain for Enhanced rapid Design, Grant Agreement Number 214889, IST-2001-34820 ARTIST and IST-004527 ARTIST2 Networks of Excellence on Embedded Systems Design.

978-3-9810801-5-5/DATE09 © 2009 EDAA

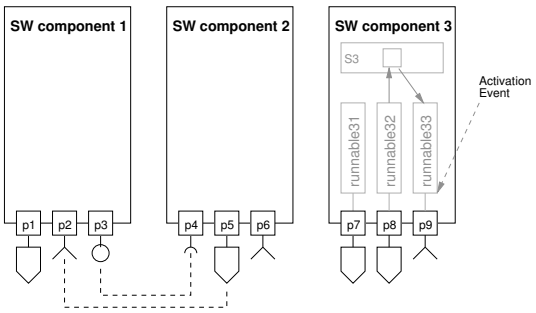

Fig. 1. Autosar components, interfaces and runnables.

a set of ports for data-oriented or service-oriented communication. In the first case, the port represents (asynchronous) access to a shared storage in which one component may write to and others may read from. In the case of service-oriented communication, a client component may invoke the services of a server component. Components may interact by matching ports carrying a request for data access of type "Receive" with ports of type "Send" and ports of type "Client" with ports of type "Server". These connections occur over the AUTOSAR Virtual Functional Bus or VFB, for which an actual implementation is then automatically generated depending on the placement of the components in the physical architecture. In our model, each component is labeled as C_i. Ports are labeled as p_i.

The *behavior* of each component is represented by a set of *runnables*, that is, procedures that can be executed in response to events, such as timer activations (for periodic runnables), or data writes or other types of application signals. Runnables of component C_i are denoted as ρ_{ij}. When reference to the component is not needed, runnables can be identified by a single index ρ_j. For each runnable we assume knowledge of its best-case execution time or β_j and its worst-case execution time γ_j.

Runnables may need to update as well as use state variables for their computations. this often requires exclusive access (write/read) to such state variables, labeled as s_{ij} (the $j-th$ variable of C_i). In this work, we restrict to runnables that are activated in response to periodic timer events or provide implementation to server functions that are called by other runnables. Therefore, we associate to each runnable ρ_j a period T_j and a worst case jitter J_j. For runnables activated directly by a periodic event, it is $J_j = 0$ and the period is the period of the activating event. For a runnable ρ_k activated in response to another runnable's request, we identify the periodic event that activated the calling runnable or, if it is not activated periodically, the one calling it and so on iteratively. If the first runnable in this chain is ρ_q, then $T_k = T_q$ and $J_k = \sum_i(\gamma_i - \beta_i)$ where the sum is computed over all the predecessors of ρ_k in the calling sequence.

Finally, the implementation of runnables consists of the code implementing the functionality, which is executed by a set of threads in a task and resource model, although in AUTOSAR the task level is, somewhat improperly, still defined as behavioral model. The implementation of runnables into tasks can be modeled as follows. $\mathcal{T} = \{\tau_1, \ldots, \tau_l\}$ is the set of *tasks*. Each task τ_i has a priority π_i and an activation period T_i. Each task is strictly periodic and starts with phase ϕ_i. A mapping relation $m(\rho_i, \tau_j, k)$ may be defined between a runnable ρ_i and a task τ_j meaning that the code implementing the runnable ρ_i is executed in the context of task τ_j with ordering index k. A mapping relation is only possible if the execution rate of ρ_i and τ_j are the same. Furthermore, runnables must be mapped in such a way that ordering relations (for example, resulting from a sequence of calls) are preserved. Although one of the main objectives of AUTOSAR is to cope with complex distributed architectures and the placement of SW components on the ECUs of a distributed system, in this paper, we only deal with timing issues at the local level, that is for components mapped into ***tasks executing on the same ECU***. In the end, the mapping of runnables into tasks takes the shape of Figure 2.

We denote by $C_{j,k}$ the worst case computation time of the task τ_j up to the k-th runnable mapped onto it and by C_j its worst case computation time. Similarly, we denote as $o_{i,j}$ the minimum offset between any two activation events of tasks τ_i and τ_j and by $O_{i,j}$ the maximum such offset.

$$C_{j,k} = \sum_i \gamma_i$$

Where i spans over all the runnables ρ_i for which the relation $m(\rho_i, \tau_j, l)$ is defined with $l \leq k$.. Similarly, we define the *worst case response time* of task τ_j (up to the k-th block) as r_j ($r_{j,k}$). The worst case response time of a task τ_j and of a runnable ρ_i executed as the k-th block of task τ_j can be respectively computed by applying the following well-known formulas [1]:

$$r_j = B_j + C_j + \sum_i \left\lceil \frac{r_j}{T_i} \right\rceil C_i \tag{1}$$

$$r_{j,k} = B_j + C_{j,k} + \sum_i \left\lceil \frac{r_{j,k}}{T_i} \right\rceil C_i \tag{2}$$

where the index i spans over higher priority tasks τ_i ($\pi_i \geq \pi_j$). the term B_j represents the worst-case blocking time for the i-th task or the k-th block mapped into it, respectively, that is, the time spent by the task or the block waiting for another task executing at a lower priority level. This blocking time is the result of blocking because of impossibility of preempting a lower priority task (if the scheduler is, even if temporarily, non-preemptive), or the result of a wait on a critical section or on another type of condition. In real-time applications, like those that are subject of this research, each runnable must complete before a deadline d_i. From the previous formulas, there are two clear indications. The response time of a runnable depends on the task into which it is mapped, on its priority (the higher priority tasks), but also on its order of execution inside the task. Also, a large value of B_j can possibly affect schedulability.

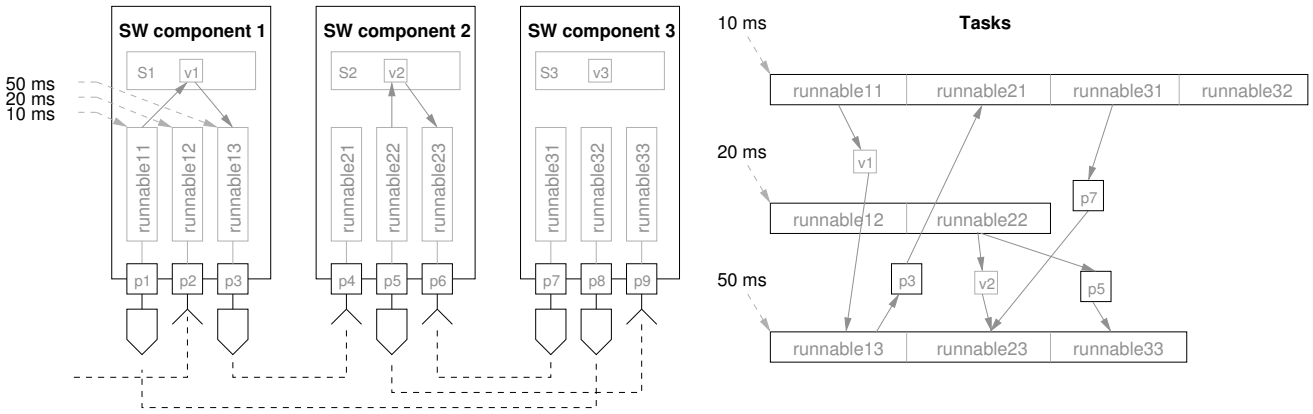

Fig. 2. Mapping of runnables into tasks.

A. Case study

Our target application consists of several tasks and runnables running at different rates. In this work, we focused on the tasks and runnables of Table I. All times are in ms.

Task	Exec time (avg)	Exec time (max)	Runnables
2ms	0.85	1.15	50
10ms	1.88	2.13	70
100ms	0.309	0.36	15

TABLE I
TASKS AND RUNNABLES FOR OUR TARGET APPLICATION.

The communication variables among the last two tasks amount to a total of 214 for a memory requirement of 595 bytes. An additional buffer applied to all of them would result in almost 600 additional bytes of RAM required. As for the running times, as shown in the table, the average execution time of a runnable is from 15 to 26 μs and the γ_i go from 23 to 30 μs. The overheads for the lock and unlock operations on shared resources by the (OSEK) OS are 0.5 μs for locking a resource and 0.9 μs for unlocking it.

III. DATA CONSISTENCY AND TIME DETERMINISM

There may be several types of consistency issues when mapping runnables into tasks

- Runnables in the same component share variables in the component's state and are activated at different rates. This case is shown in the left hand-side of Figure 3. In this case, we must ensure access to the state variables is performed in a mutually exclusive way by the runnables. Runnable ρ_{11} activated with a rate of $T = 10ms$ is writing information that is read by ρ_{13} executed every 20ms. We assume runnables are mapped into two tasks executing at their rates and the high rate task has higher priority. In this case, the task implementing ρ_{11} can preempt the task implementing ρ_{13} while the latter is updating the state variables, thereby using an inconsistent version of the component's state. *In this case, we need to enforce the impossibility of mutual preemption, at least during the time needed to access the state variables.*
- One runnable makes use of state variables and needs to be executed in response to multiple events at different rates or multiple asynchronous events. In this case, state variables can be accesses by multiple instances of the runnable and they need to be protected against concurrent read-write access. This is a special version of the previous case.
- There is (data-oriented) communication using ports with non-atomic data between runnables activated at different rates. One such scenario is represented in figure 3. Runnable ρ_{13} is activated with a rate of $T = 20ms$ and writes information that is read by ρ_{21} executed every 10ms (oversampling). Once again, we assume runnables are mapped into two tasks executing at their rates and the high rate task (the receiver in this case) has higher priority. In case runnables are communicating on a data port representing a type not accessed in an atomic way, there can be data consistency issues if ρ_{13} is preempted by the task of ρ_{21} while it is updating the communication variables. In this case there may be a consistency issue when part of the variables are written by the current instance of ρ_{13} and part by the following instance while possibly being interrupted in between by ρ_{21}, but also a time determinism issue since ρ_{21} can use data produced by any of the two instances of ρ_{13}. Data consistency should also be required, and once again, this means

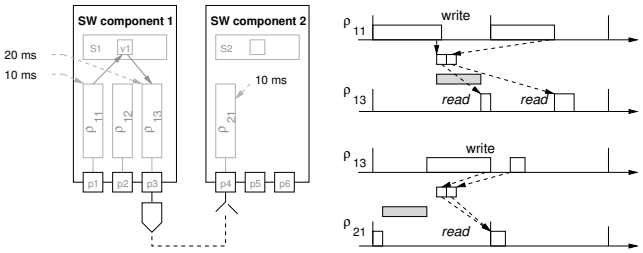

Fig. 3. Data communication among runnables at different rates.

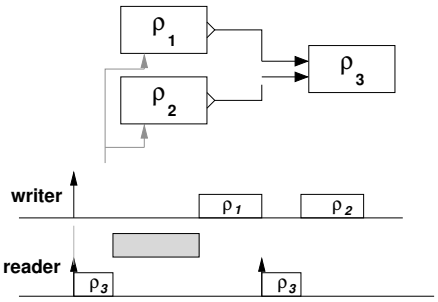

Fig. 4. Time consistency issues with respect to multiple inputs.

ensuring access to the communication variables in a mutually exclusive way by the runnables. However, also time determinism should be provided to ease verification of the system properties. In this case, we need to ensure that ρ_{21} always uses the values produced by the first instance of ρ_{13} and additional mechanisms are needed like the Rate Transition block described below.

- Runnables operate on multiple inputs coming from runnables at different rates. In this case, the reader may require that all inputs are produced by instances of the writer runnables activated at the same time. This form of time determinism on the input data may allow better control or predictability over the functioning of the reader runnables.
  Figure 4 shows a possible counterexample, in which runnable ρ_3, executing at higher rate/priority, reads the latest value produced by runnable ρ_1 and the value produced by the previous instance of ρ_2. In this case, to ensure this type of time determinism, it is necessary to *prevent the reader runnable from executing between the updates of the communication variables* by any of the writers belonging to the same instance set. A simpler solution would be to consider all the writer runnables as a non-preemptable set by the reader.

IV. Providing data consistency and time determinism

There are several ways to make sure that the runnable implementation into tasks does not incur in any of the previous problems. Before discussing the possible approaches, however

a) Ensuring there is no preemption by time analysis: The first option consists in using time analysis to demonstrate the impossibility of mutual preemption between writer and reader (this also ensures time determinism). These conditions can be summarized by the following cases.

A high priority writer runnable ρ_w will not preempt a low priority reader runnable ρ_r if

$$r_w \leq T_w - O_{wr}$$

where O_{wr} is the maximum offset between the activation of the writer and the corresponding reader instance. If writer and reader are harmonic and activated by the same event, then

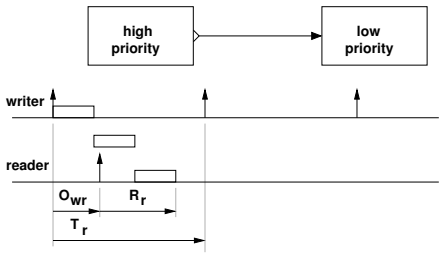

Fig. 5. Conditions for absence of preeemption between a high priority and a low priority task.

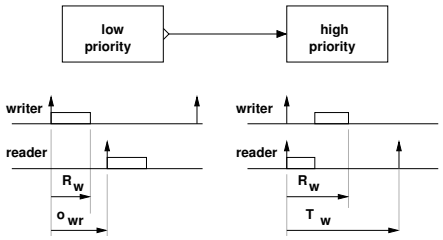

Fig. 6. Conditions for absence of preeemption between a low priority and a high priority task.

$O_{wr} = 0$. In the case of a low priority reader and high priority writer (typical of oversampling), the condition is

$$o_{wr} > r_w \vee (o_{wr} = O_{wr} = 0 \wedge r_w \leq T_r)$$

where the second condition is the only one with practical applicability.

When this type of guarantee is possible, there is clearly no additional cost for the implementation. Note, however, that the previous constraints depend on the relative phases of reader and writers and on the completion time of the reader and writer runnable (or the runnable accessing the state information). This completion time depends on how the runnable is mapped inside the task (Equation 2).

b) Disabling preemption among runnables: The second option consists in simply preventing preemption while runnables are executing and allowing it at the boundaries between runnables. This option has minimum impact on the code and results in an additional blocking time when computing the schedulability of the tasks. The worst case blocking time is the duration of the longest runnable. From the memory standpoint, it does not require any additional memory for buffering mechanisms. The major practical consequence is the delaying of the worst case response time of runnables. In this case, the $r_{i,k}$ of each runnable is computed considering a term B_i that is equal to the maximum of the terms γ_j where the runnables to be considered are those mapped into a task with priority lower than the priority of ρ_i.

c) Wait free methods (Rate transition blocks): When the execution time of each runnable is significant if compared with the execution time of a task, the previous approach can result in excessive blocking and a different approach is required. The first alternative consists in protecting the state or communication variables using wait-free methods. A wait-

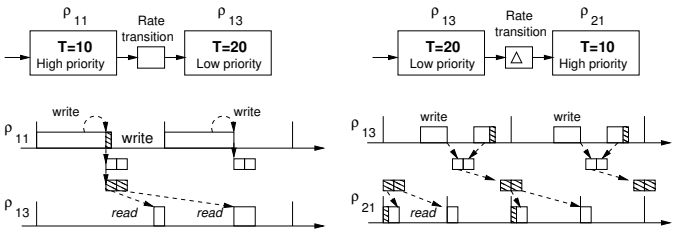

Fig. 7. Rate transition blocks: a special case of wait-free buffering scheme.

free approach for protecting communication buffers is the Rate Transition buffer approach of the code generators by Mathworks [6].

The Rate Transition block behaves like a Zero-Order Hold block (for fast to slow transitions) or a Unit Delay block plus a Hold block or Sample and Hold (for slow to fast transitions). Its implementation consists of a switched buffer.

Conceptually, in the fast to slow (high to low priority) case, the RT block output update function executes at the rate of the slower block, but with the priority of the faster block (dashed box in the left side of Figure 7.) In low to high rate (priority) transitions, the RT block output update function runs in the context of the fast task, but at the rate of the slow task as the first function (dashed box in the bottom-right side of the Figure) and it feeds the runnables of the fast task. The RT state update function executes in the context of the slow task as the last function and update the internal buffer state (the striped box in the right side of the Figure.) The output function uses the state of the RT that was updated in the previous instance of the slow task. In the case of synchronous activation of communicating tasks (when $o_{wr} = O_{wr} = 0$, the RT block controls the timing of data transfer in a completely predictable way, avoiding data consistency issues and guaranteeing time determinism. In the cases in which activations are not synchronous, different mechanisms can be used [7], [2], [3].

The cost of these mechanisms is additional memory for the implementation of the communication buffers. For the case of Rate transition blocks, this memory is an additional set of output variables for transitions from high to low rate (priority) and an additional set pair of state and output variables for transitions from low to high rate (priority). The mechanisms require also some timing overhead for the management of the communication buffers. This timing overhead is typically included in the computation time of the runnables.

d) Semaphore locks: The other possibility is the use of immediate priority ceiling semaphores (or OSEK resources). In this case, access to the state and communication variables is protected by lock and unlock primitives (get and release resource). The impact of this solution on the memory and time characteristics of the application is the following. On the memory side, there is no need for replicating any state or communication variable. However, the implementation of the semaphores require a minimum amount of additional memory, even when they are immediate priority ceiling semaphores and the only required action is to raise the priority of a task (runnable) to the ceiling priority of the resource whenever it starts using it. From the point of view of the timing properties, these mechanisms result in a contribution to the blocking term B_i equal to the longest critical section used for accessing communication or state variables by lower priority runnables/tasks. This value may be much smaller than the execution time of the entire runnable and bring a significant advantage with respect to the case of preventing preemption among runnables.

V. Porting legacy code

When runnables are implemented by porting legacy code, an additional requirement must be considered. The mechanisms for data consistency and flow preservation should be implemented with minimum changes to the existing code. Clearly, the proposed methods require different approaches and may have widely different costs. Disabling preemption at the beginning of runnables and enabling it again at the end has requires very limited changes (one or two lines of code at the beginning and at the end of each runnable. Also, the implementation of wait-free buffering mechanisms can be implemented with very little changes in many cases, by simply adding an additional header file with a few macros to the code implementing the reader runnables and adding a small portion of code at the end of the writer runnable. Adding the code for getting and releasing an (immediate priority ceiling) OSEK (or AUTOSAR OS) resource before and after the use of each set of communication or state variables is however much more difficult and would require multiple changes.

VI. Optimization opportunities

As stated in the previous section, several methods can be used to ensure data consistency and time determinism. In some cases, it is possible to achieve safe operation without additional costs, by simply demonstrating absence of preemption. This sometimes requires careful selection of the mapping order of runnables into tasks. In the other cases, several methods are available that offer tradeoffs between the required amount of additional memory and the runtime costs of operating systems mechanisms versus the amount of additional blocking time that is imposed over tasks. These methods can be used in a combination, for example, by executing first inside tasks runnables with large sets of communication and shared state variables, using time analysis to guarantee their consistency, and then by preventing preemption among runnables, when those runnables are very short, leaving the other mechanisms to the remaining cases. By formalizing the memory cost and the amount of additional execution time or blocking time that is required by the following methods, it should be possible to define an optimization problem as follows.

Define Runnables mapping and protection mechanisms to ***Minimize*** Memory requirements,

Subject to Timing constraints

and to formally look for the memory optimal solution to the problem.

VII. Conclusions

We described some of the issues in the implementation of a set of AUTOSAR runnables providing the functional behavior of an automatic gear application into a set of concurrent tasks. The requirements for the consistency of communication and state variables and the possible additional requirements of flow preservation (time determinism on the communication) can be satisfied using several protection mechanisms. The available methods offer tradeoffs between time response (and time overheads) and demand for additional (RAM) memory. Memory costs need to be analyzed, in light of the time constraints of the application, to select the best mechanism for the application runnables or possibly a combination of them within the same application.

Acknowledgment

The author wish to thank Felice Tufo and Nicola Ariotti from Magneti Marelli S.p.A. for their insightful discussions and contributions to the work.

References

[1] M. G. Harbour, M. Klein, and J. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," IEEE Transactions on Software Engineering, vol. 20, no. 1, January 1994.

[2] Baleani, M., Ferrari, A., Mangeruca, L., and Vincentelli, A. S. 2005. Efficient embedded software design with synchronous models. In *Proceedings of the 5th ACM EMSOFT conference*.

[3] M. Di Natale, G. Wang and A. Sangiovanni-Vincentelli, Optimizing the Implementation of Communication in Synchronous Reactive Models, In *IEEE Real Time Applications Symposium*, St Louis, April 2008

[4] OSEK. Osek os version 2.2.3 specification. available at http://www.osek-vdx.org, 2006.

[5] W. Nesci and S. Angellotti. AUTOSAR: A Global Standard - Magneti Marelli System and Application development with AUTOSAR. 1st AUTOSAR Open Conference, Detroit, October 23th , 2008.

[6] Mathworks. *The Mathworks Simulink and StateFlow User's Manuals*. Mathworks. web page: http://www.mathworks.com.

[7] N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *6th Euromicro ECRTS*, July 2004.

[8] Tripakis, S., Sofronis, C., Scaife, N., and Caspi, P. 2005. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers. *Proceedings of the 5th ACM EMSOFT conference*.

[9] Marco Di Natale and Valerio Pappalardo. Optimizing the multitask implementation of multirate Simulink modelss. *ACM Transactions on Embedded Systems*.

[10] S. Boyd and L. Vandenberghe. Convex optimization. Available at http://www.stanford.edu/boyd/cvxbook/, 2004.