Undoing the Task: Moving Timing Analysis back to Functional Models

Marco Di Natale
Scuola Superiore S. Anna, email: marco@sssup.it

Haibo Zeng
McGill University, email: haibo.zeng@mcgill.ca

The real-time systems community has traditionally considered tasks or jobs (from the operating system concept of thread) as the units for the analysis model. With time, more complex task models have been created to represent conditional execution (branching), precedence constraints, and an increasingly complex model of time dependencies, from the multiframe model until the most recent extended digraph model. In the past, research works have explored the benefits of breaking the task structure to enforce a different management for subsets of the task execution time (for example, by restricting preemption, or changing the execution priority). Examples are the dual priority scheduling [3] and the non-preemption sections at the end of the task [4].

In the meantime, the industrial world is moving away from the traditional manual programming to adopt model-based design. The threads (as concurrent units of execution, managed by the operating system) are in the background, and functional models, such as dataflows or networks of synchronous blocks, including extended finite state machines are the modeling entities. The task (or threads) model becomes an intermediate artifact, and the timing analysis becomes part of a synthesis problem. The problem constraints are the semantic properties of the functional model that need to be preserved, and the task model must guarantee a correct implementation that is feasible and memory effective or time-robust.

I. Functional models

Most functional models in use today are built on a synchronous reactive (SR) semantics. For simplicity, we restrict to discrete-time models. The system is a network of functional blocks b_j. Blocks can be of two types. Regular blocks process a set of (discrete time) input signals at times that are multiples of a period T_j, which is in turn an integer multiple of a system-wide *base period* T_b (the model could be extended to sporadic activations). We denote inputs of block b_j by $i_{j,p}$ ($\overline{i_j}$ as vector) and outputs by $o_{j,q}$. At all times kT_j the block reads the signal values on its inputs and computes two functions: an output update function $\overline{o_j} = f_o(\overline{i_j}, S_j)$ and a state update function $S_j^{\text{New}} = f_s(\overline{i_j}, S_j)$, where S_j (S_j^{New}) is the current (next) state of b_j. Often, the two update functions can be considered as one $\overline{o_j}$, $S_j^{\text{New}} = f_u(\overline{i_j}, S_j)$. For timing analysis, the worst execution time of the update function is estimated as γ_j.

State machine (SM) blocks can have multiple activation events $e_{j,v}$ (as shown in the right hand side of Figure 1 with two events of period 2 and 5). At any integer multiple of one of the events' periods $kT_{j,v}$, an update function is computed depending on the current state, the subset of input events that are active and the set of input values. Update functions are typically extended by allowing the execution of generic functions whenever a given event is active on a given state. When multiple events are active, an order is provided to give some events (for the given state) precedence over others (thereby guaranteeing determinism). This is typically summarized in a graph representation as in the right side of Figure 1. The figure represents an SM with two events with periods 2 and 5 and the corresponding possible activation times and actions.

In the case of a state machine block, it pays off to identify the worst-case execution time associated to each update/action for each state, event and set of input values. The structure of the state machine constrains which update/actions can occur in the worst case within a given time interval (for details refer to [2]). The procedure to calculate the request and demand bound functions for state machines is similar to those used for digraph task models. For the example of Figure 1, the worst-case sequence of actions is defined by the state graph (which transitions are possible out of which state). Also, a trivial solution consists in an implementation with a single task running at the greatest common divisor of the events periods, but different task models may be defined.

If two blocks b_i and b_j are in an input-output relationship (one of the outputs of b_i is the input of b_j, and the output of b_j depends on its input), there is a communication link between them, denoted by $b_i \rightarrow b_j$. Let $b_i(k)$ represent the k-th occurrence of block b_i (belonging to the set of time instants $\bigcup_v kT_{i,v}$ if a state machine block, or kT_i if a standard block), then a sequence of activation times $a_i(k)$ is associated to b_i. Given $t \geq 0$, we define $n_i(t)$ to be the number of times that b_i has been activated before or at t.

In case of a link $b_i \rightarrow b_j$, if $i_j(k)$ denotes the input of the k-th occurrence of b_j, then the SR semantics specify that this input is equal to the output of the last occurrence of b_i that is no later than the k-th occurrence of b_j, i.e., $i_j(k) = o_i(m)$, where $m = n_i(a_j(k))$. This implies a partial order in the execution of the block functions (different from the precedence constraints assumed in task models). The

top timeline on the left of Figure 1 illustrates the execution of a pair of blocks with SR semantics. The horizontal axis represents time. The vertical arrows capture the time instants when the blocks are activated and compute their outputs from the input values. In the figure, it is $i_j(k) = o_i(m)$.

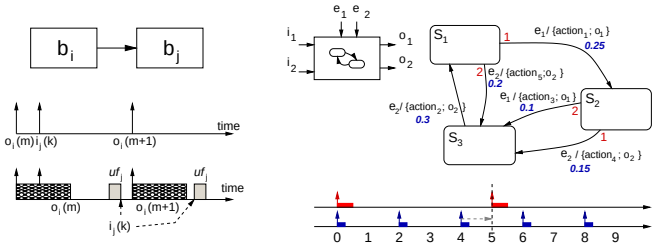

Figure 1. input-output relationship among blocks (left) / state machine and activation events (right)

The update functions and their action extensions are executed by program functions (or lines of code) executed by a task, thereby providing the level of granularity that can be leveraged to improve schedulability (for example, by disabling preemption). This makes the task structure a design artifact or objective, rather than the starting point.

The function-to-task mapping consists of a relation between a block update function (or each one of them in the case of an FSM block) and a task, and a static scheduling (execution order) of the function code inside the task. The i-th task is denoted as τ_i. $\mathcal{M}(f_{i,k}, p, n)$ indicates that the function $f_{i,k}$ of block b_i is executed as the n-th segment of code in the context of τ_p.

II. The synthesis problem

The stage of the design process in which the functional model is mapped into a task (thread) model is the starting point of several optimization problems, including how to map functions into tasks, how to assign the execution order of functions inside tasks, how to assign the task parameters (priority, deadline, offset) to guarantee semantics preservation and schedulability, how to assign scheduling attributes to functions (including preemptability and preemption threshold) and even how to design communication mechanisms that ensure flow preservation while minimizing the amount of memory used.

The bottom-left side of Figure 1 shows the possible problems with flow preservations in multi-task implementations. The writer finishes its execution producing $o_i(m)$. If the reader performs its read operation before the preemption by the next writer instance, then (correctly) $i_j(k) = o_i(m)$. Otherwise, it is preempted and a new writer instance produces $o_i(m+1)$. In case the read scheduling is delayed, the reader reads $o_i(m+1)$, in general different from $o_i(m)$.

The correct set of values may be provided to the reader by enforcing an execution order by the scheduler or by using a suitable communication mechanism (such as an instance of wait-free communication), with the associated memory overhead.

The mapping of functional blocks into tasks, the configuration of the task model, and the selection of the mechanisms for the implementation of the communication over ports (protecting against data inconsistency and possibly flow semantics violations) have a large impact on the performance of the system. The selection of the communication mechanism and the protocol to protect state variables leverages tradeoffs between time overhead for the execution of the protocol, memory required for the implementation of the mechanism, and possible blocking time. For implementation on single-CPU architecture platforms, solutions have been proposed (not exhaustively, many problems are still open).

Examples of problems that are open are the following: Given a system composed of multiple communicating state machine blocks and dataflow blocks to be executed onto a multicore platform, find the mapping of the state machine actions and block reactions onto a suitable set of tasks, the placement of such tasks onto the cores and the assignment of activation offsets and priorities to tasks such that the partial order of execution defined by the functional model semantics is preserved and each block processes its inputs and computes its next state and output in time for the next execution of the follower blocks. As starting point, a discussion on possible task implementations for a single state machine block is provided in [1] and the time analysis of finite state machine actions, implemented by a single task (with similarities to the analysis of generalized digraph models [5]) is discussed in [2].

However, other (possibly) simpler problems also exist. one example is the following: define the set of tasks that can provide an implementation to a network of block actions in a multicore platform, with the assignment of task priorities and possibly activation offsets, and the assignment of preemption thresholds to actions inside the tasks (to limit preemptability) in such a way that the implementation is correct and the use of memory (for stack and communication) is minimized.

References

[1] M. Di Natale and H. Zeng, "Task implementation of synchronous finite state machines," in *Proc. the Conference on Design, Automation, and Test in Europe*, 2012.

[2] H. Zeng and M. Di Natale, "Schedulability Analysis of Periodic Tasks Implementing Synchronous Finite State Machines," to appear in *Proc. 23rd Euromicro Conference on Real-Time Systems*, 2012.

[3] R. Davis and A. Wellings, "Dual priority scheduling," in *Proc. the 16th IEEE Real-Time Systems Symposium*, 1995.

[4] M. Bertogna, G. Buttazzo, and G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions", in *Proc. the 32th IEEE Real-Time Systems Symposium*, 2011.

[5] M. Stigge, P. Ekberg, N. Guan, and W. Yi, "The digraph real-time task model," in *Proc. 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.