Task Implementation of Synchronous Finite State Machines

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

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

***Abstract*—Model-based design of embedded control systems using Synchronous Reactive (SR) models is among the best practices for software development in the automotive and aeronautics industry. SR models allow to formally verify the correctness of the design and to automatically generate the implementation code. This improves productivity and, more importantly, can ensure a correct software implementation (preserving the model semantics). Previous research focuses on the concurrent implementation of the dataflow part of SR models, including the optimization of the block-to-task mapping and communication buffer sizing. When the system also consists of blocks implementing finite state machines, as in modern modeling tools like Simulink and SCADE, the task implementation can be further optimized with respect to time and memory. In this paper we analyze problems and opportunities in the implementation of finite state machine subsystems. We define the constraints and efficient policies for the task implementation of such systems.**

I. Introduction

The development of complex embedded systems is subject to tight cost and performance constraints. Automatic code generation tools producing a software implementation of an application model, defined according to a high-level (possibly visual) language are being adopted to increase productivity and avoid errors in the development of embedded software.

In the development of embedded controllers, the use of the Simulink visual language and modeling tool is becoming widespread, together with the associated code generators such as Real-Time Workshop (RTW), Embedded Coder (EC) from MathWorks [9] and TargetLink of dSPACE [7]. The market relevance of Simulink is such that rules and automatic translation tools have been developed for converting a Simulink diagram into an equivalent Lustre [11] or Esterel [2] description for the purpose of formal verification of properties and/or provably correct code generation. Commercial products for the verification of properties of Simulink models and automatic test generation with guaranteed coverage are also available [8].

A Simulink model is a network of blocks. Each block processes a set of input signals and produces a set of output signals. All Simulink blocks, when executed, compute two functions: the *state update* function (possibly omitted in purely functional blocks), which updates the next block state based on the current state and the values of the input signals, and the *output update* function, computing the set of values for the current time for the output signals as a function of the current state and the inputs. Whenever the block outputs depend on the block input values, the block must execute after the predecessor blocks. This introduces a partial order in the execution of the block functions.

Although Stateflow (and Simulink) allows for continuous-time signals and events occurring at arbitrary times, in this paper we are interested in ***the subset of the Simulink/Stateflow language to which automatic code generators currently apply***. In this case, continuous signals must be realized using a discrete time solver, and all signals and events in the system are discrete-time functions defined over the multiples of a base period for the system model (the solver period). Likewise, we do not consider blocks activated by function calls.

Simulink blocks are of two types. One type, somewhat improperly called *Dataflow*, is executed at a given period (integer multiple of the *base period*). For these blocks, the complexity of the output update and state update functions does not depend significantly on the block state (which typically appears only as a parameter for those functions). For the other type of blocks, an explicit modeling of the states and the dependency of the update functions on the state is required. These blocks can be activated by signals coming at multiple rates. They are explicitly defined as (Mealy type) *Extended Finite State Machines*, or extended FSMs. In Simulink, they are called *Stateflow* blocks. In Stateflow blocks, each trigger event may also result in the execution of a set of *actions*. Stateflow A Stateflow block receives as input a set of signals and a set of events obtained from signals. As a result of its reaction, the block updates a set of output signals. The events that trigger the reaction of the block (if no trigger event is specified, the block reacts at thel base rate) are obtained from periodic signals and are therefore assumed to be *periodic*.

While implementing a Simulink model into code, the problem is to provide a feasible implementation (for example, with respect to time and memory constraints) that preserves the logical-time execution semantics (the rate and order of execution of blocks) and the communication flows.

Current commercial code generators provide correct implementations of Simulink models for single-processor CPUs at the price of possible pessimism with respect to schedulability. In a single-task implementation, all blocks are executed by one task running at the system base period according to a global order that is compliant with their partial order execution constraints. Feasibility requires that the longest reaction in the system completes before the task is executed again (the task deadline is equal to the *base period*).

Multitask implementations are desirable because they im-

978-3-9810801-8-6/DATE12/©2012 EDAA

prove schedulability. All the blocks with the same rate are executed by the same task and tasks are scheduled by priority with a *Rate Monotonic* policy. A multitask implementation brings issues because of the need to guarantee the data consistency of the signal flows that occur between blocks executed by different tasks. The Simulink code generators provide Rate Transition (RT) blocks to guarantee the consistency of shared data, and the preservation of communication flows (time determinism) whenever there is a communication between two blocks (tasks) with different rates. The RT block behaves like a Zero-Order Hold block for transitions from a high-rate to a low-rate block. The block output values are updated by an output update function executed in the context of the writer task, after the completion of the writer block, but with the rate of the reader block (Figure 1). In the case of low-to-high rate transitions, the RT block behaves like a Unit Delay block plus a Hold block. In this case, the reader executes before the writer (because of the Rate Monotonic policy), hence a functional delay. The RT block state update part executes in the context of the writer task, right after the writer block, while the output update part executes in the context of the reader task, before the reader block, with the period of the writer.

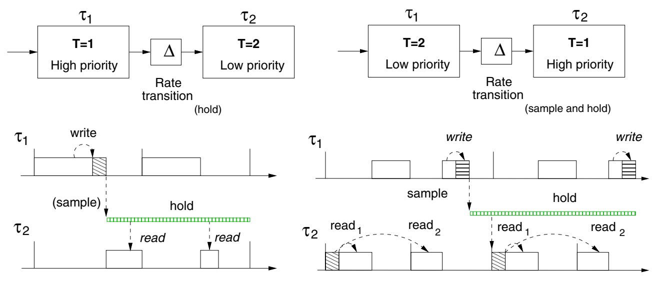


Fig. 1. Rate Transition blocks are added between blocks executing at different rates to guarantee the preservation of the communication flows.

RT blocks for high-to-low rate transitions require an additional set of output variables for communication between the sender and the receiver as well as (limited) additional code for the output update function of the block. The memory overhead is equal to the size of the variables implementing the communication link. RT blocks for low-to-high rate transitions require additional sets of state and output variables as well as the corresponding additional code for the state update and the output update functions. The memory overhead is double the size of the communication link. In addition, this type of RT block results in an additional functional delay equal to the period of the slower block. The introduction of zero-order hold blocks and time delays improves schedulability at the price of additional memory, and possibly a performance degradation of the feedback control system because of the additional delay.

In the multi-task implementation of Simulink models, the Embedded Coder/Real-Time Workshop [9] tools assumes a single periodic task implementation for each Stateflow block code. Each time the task is activated, it checks for any active trigger and, if there is any, it processes them. Each Stateflow block therefore executes at the greatest common divider (gcd) of its trigger signals. Such a model has several limitations:

- It forces the execution of all transitions in a single task, with the same priority level, short period (and typically high priority), and short deadlines. Therefore it imposes a high interference on the other tasks with lower priority.
- The execution of the task at the *gcd* of the period of the trigger events (a possibly very short period) makes very likely the addition of RT blocks at its input/output links.

A multi-task implementation of a Stateflow block can provide more flexibility and efficiency in terms of real-time schedulability. Moreover, it can potentially avoid the addition of RT blocks in the data transfer between tasks with different rates, thus saving on the associated memory overhead. In this work, we propose semantics-preserving efficient implementations of Stateflow blocks activated by triggers with different rates using multiple concurrent tasks scheduled by priority. *We define several options for time- and memory-efficient multi-task implementations of a synchronous FSM* and we analyze their benefits in terms of schedulability and memory usage.

The code implementation of synchronous state machines has been discussed at length by Berry and Gonthier [3], and Benveniste et al [1]. The scheduling problem needs to account for the partial order in the execution of blocks required by the Mealy semantics of the state machines and also for the need to complete the system reaction by the time a new event arrives in the system. However, most of the discussed implementations consist of static scheduling of code in a single task implementation. For multitask implementations previous works focused mostly on the implementation of the intertask communication [4], rather than the real-time analysis or the synthesis of an efficient task structure. The possible avoidance of RT blocks by the selection of a different block-to-task mapping and priority assignment model has been proposed in [6] for the case of Dataflow blocks only. In this work, we assume the use of RT blocks at every rate transition. However, the two methods could be combined to produce an even more efficient implementation. We plan to explore this opportunity in future work.

The paper is organized as follows. Section II summarizes the model of synchronous Finite State Machines. Section III proposes the possible multitask implementations, with examples in Section IV. Section V gives the experimental results on potential memory savings using random task graphs. Finally, Section VI concludes the paper and discusses future work.

II. Synchronous FSM abstract model

A synchronous model consists of a graph of communicating Mealy Finite State Machines (FSMs). Each FSM is defined by a tuple $(\mathbf{S}, S_0, \mathbf{I}, \mathbf{O}, \mathbf{E}, \mathbf{T})$, where $\mathbf{S} = \{S_0, S_1, S_2, \ldots S_q\}$ is a set of *states*, $S_0 \in \mathbf{S}$ is the initial state, $\mathbf{I} = \{i_1, i_2, \ldots i_n\}$ and $\mathbf{O} = \{o_1, o_2, \ldots o_p\}$ are the *input* and *output* values, where each i_j (o_j) is a signal, also denoted as s_j. Each signal is a function defined on a discrete time domain and with values in a given range. The discrete time domain of each signal s_j matches the system *base period* t. Signal values only change at multiples of its period, defined as $t_j = k_j \cdot t$, and are persistent between updates. $\mathbf{E}$ is the set of *trigger (or activation) events*. Each event e_j is generated by a (rising or falling) value transition of a signal and is therefore bound

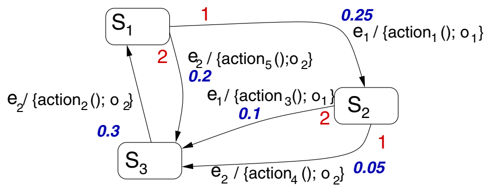


Fig. 2. An example of FSM behavior description

to occur only at time instants belonging to a time base with period t_{e_j} which is an integer multiple of t. At each time $t_{e_j} = k_{e_j} \cdot t$ the event may or may not be present (alternatively, the machine may stutter). The periods of all signals and events are therefore multiple of the *base period*. $\mathbf{T}$ is the set of transition rules. Each transition $\theta_j \in \mathbf{T}$ consists of a tuple $\theta_j = \{S_{s_j}, S_{d_j}, e_j, g_j, a_j, p_j\}$, where S_{s_j} is the source state, S_{d_j} is the destination state, $e_j \in \mathbf{E}$ is the trigger event, g_j is the guard condition: an expression of the input/output signals (the model may be extended to allow also the definition of internal variables), a_j is the *action*, and p_j is the transition priority (the lower the number, the higher the priority). Figure 2 shows an example of the graphical notation that is used to describe the states and transitions, along with the event, guard and action associated with each transition. The execution time (in bold, below the action) and priority (near the starting point) of each transition are also denoted in Figure 2.

In this paper we do not consider the case of transitions activated by a logical expression evaluated on multiple events. Our FSM model and task models could be easily extended to deal with this case (using a set of dummy events resulting from the evaluation of the expressions), but this would unnecessarily complicate the discussion. Also, compared with Stateflow, we do not allow events to be generated by transitions.

Following the original Statecharts specification, from which it is derived, the Stateflow semantics also allow *concurrent states*, *superstates*, *entry* actions, *exit* actions, *while* actions, *join transitions* and other constructs. For a discussion of these constructs and the conditions on which some of them may be semantically equivalent, and translated to the definitions in standard (flat) extended FSM, please refer to [12].

In synchronous FSMs, all events occur with periods that are multiple of the base period and with the same phase. Therefore, *sets of events arrive at exactly the same time* (hence the name **synchronous FSMs**). Also, every reaction occurs in *logical zero time*, that is, it must satisfy the synchronous assumption, where the reaction of a network of (possibly FSM) blocks completes before the next event is processed. Priorities associated with transitions are used to discriminate which transition should be performed when two or more events arrive synchronously and two or more corresponding transitions can be taken out of the same source state.

When an FSM is translated into code, the implementation must define for each possible state and input signal the *output update* function with the corresponding *actions* and the *state update* function (the two are sometimes merged).

III. Task Implementation of the FSM

When a synchronous FSM is translated into code, the model properties should be preserved to retain the results of the simulation and validation performed on the model. In the case of synchronous FSMs, we want to preserve the correspondence between input and output stream value, also called *flow preservation*. A task implementation is therefore *correct* as long as it preserves the set of output values that are produced in correspondence to a given set of inputs. This requires that the state and the outputs are updated in a timely fashion. The state needs to be updated before the next set of input values is processed (by the task activated by the next event). Similarly, outputs need to be produced without overwrites and in time for the following blocks to use them. This last requirement is handled by an appropriate definition of the priority levels of the tasks implementing the FSM block and its successors, plus the possible use of RT blocks.

The set of tasks in our model is denoted as $\mathcal{T} = \{\tau_1, \tau_2, \ldots \tau_m\}$. If a transition θ_j (and the corresponding action a_j) is mapped into τ_i, we write $\mu(\theta_j) = \tau_i$. The task implementing θ_j is also indicated as $\tau(\theta_j)$. The period of τ_j is indicated as ψ_j and its priority as π_j.

The Real-Time Workshop/Embedded Coder code generator adopts a single task implementation for the reactions of a Stateflow block. The code implementing the FSM *output update* and *state update* functions executes in a task with period equal to the gcd of the periods of the trigger events. We call this implementation as the *baseline implementation*, since it is simple and applies to all synchronous FSMs.

Baseline model

Formally, in a baseline implementation, a single task τ_1 implements all transitions. $(\forall j,\ \mu(\theta_j) = \tau_1, \psi_1 = gcd_k\{t_{e_k}\})$. Every time it is activated, the task code checks the current state and then, in the order of their priority, it checks if for any of its outgoing transitions, there is an active event. If it finds one, it executes the corresponding action, updates the outputs and state and completes before it is activated again.

As shown in the following subsections, several possible multitask implementations exist. *A multi-task implementation gives at least as much design freedom as a single-task implementation on the block-to-task mapping and task priority assignment*. In addition, a multi-task implementation can possibly impose a looser deadline on some of the task instances, which further improves schedulability. The advantages on memory requirements come from the possibility of avoiding the use of RT blocks.

Before discussing the possible multitask implementations, we first identify a subset of all synchronous FSMs where *the priorities of the transitions are determined by their trigger events*. In this class of FSMs, whenever there are multiple outgoing transitions from a given state, the priority of the transitions can be uniquely determined by the events/signals associated with them. That is, if for a given state, the transition associated with event e_i has priority higher than the transition associated with e_j, and e_i and e_j can occur at the same time, then for any possible state, the transition associated with e_i must always have priority higher than the transition associated with e_j. *This multitask model (called partitioned model) applies to this subset of FSMs only, but the other two apply to any FSM.*

Partitioned model

In this model, the FSM is implemented by one periodic task for each event (period). The tasks implementing the FSM are activated synchronously (with offset = 0) and share a variable encoding the FSM state. Each task handles the transitions associated with the given event and is executed at the period of the event. Also, the priorities assigned to tasks are in agreement with the priority order of the corresponding events. In addition, an *inhibition signal* is defined for any task pair $\tau_i \rightarrow \tau_j$ whenever the event handled by τ_i has a priority higher than the one handled by τ_j. τ_i sends an inhibition signal to all lower priority tasks τ_j if one of the transitions it implements is actually executed. When this happens, τ_j skips its execution.

Deadlines can be assigned based on the requirement that outputs and state are updated before the next event is processed (with the corresponding next set of inputs). However, we only need to worry about the following events that are handled by a task with higher priority. For all the others, since they arrive later and are implemented by tasks with lower priority (tasks never block because of the use of RT blocks), the requirement will be automatically met due to the synchronous activation of tasks and the priority order. In general, the absolute deadline of a task instance is defined as the minimum value between the end of the task period and the earliest activation time of higher priority tasks implementing transitions of the same FSM.

With this definition of task deadlines, and assuming schedulability analysis can guarantee that tasks always execute within them, access to the state variable does not require any protection mechanism to ensure data consistency. Indeed, it is easy to demonstrate that preemption between tasks is not possible as long as task deadlines are met.

The partitioned model implementation preserves the FSM flows. Whatever is the state of the FSM, only one transition for one event is possibly executed and the transition priority order is preserved because tasks are activated with the same offset and higher priority tasks execute first. In addition, because of the inhibition signal, only the highest priority transition (belonging to the highest priority task) is executed. Outputs and state are updated before the next set of inputs relevant to the FSM is evaluated. This assumption that tasks complete before their deadlines requires the availability of a schedulability analysis method. Several methods could be used for a (possibly pessimistic) analysis of this type of tasks, e.g., [10]. Currently, we are investigating a comprehensive analysis method that improves the currently available algorithms.

The benefits of this implementation are multi-fold. With multiple tasks, it is no longer necessary to give the same scheduling priority to all (transition) actions, improving scheduling flexibility. Also, the deadlines of most task instances are relaxed. Finally, communication links may occur without rate transitions and therefore avoiding the use of RT blocks as shown in the examples and the experimental section. We believe in most cases this model is the most efficient with respect to ease/flexibility of scheduling and memory requirements for RT blocks. However, it cannot be always applied. The next two task models apply to any FSM.

Mixed-partitioned model

The next model is a *mixed partitioned* model. In this case, the FSM transitions are partitioned into two sets $\mathbf{T_p}$ and $\mathbf{T_b}$. The first set $\mathbf{T_p}$ includes all transitions for which, in every state of the FSM, their relative priorities are determined by their trigger events. Also, all the transitions in this first set must have priority lower than all the transitions belonging to the set $\mathbf{T_b}$ that have the same source state.

Such a partition can always be computed (if we include the case in which the first set is empty) using Algorithm 1.

Algorithm 1 Calculate the sets $\mathbf{T_p}$ and $\mathbf{T_b}$

```
1:  $\mathbf{T_p} = \mathbf{T}$
2:  $\mathbf{T_b} = \{\}$
3:  for all $S_i, S_j \in \mathbf{S}$ do
4:      for all $\theta_l, \theta_m \in \mathbf{T_p}$, with $S_{s_l} = S_{s_m} = S_i \wedge p_l > p_m$ do
5:          if $\exists \theta_v, \theta_w \in \mathbf{T}$ such that $S_{s_v} = S_{s_w} = S_j \wedge e_{t_v} = e_{t_l} \wedge e_{t_w} = e_{t_m} \wedge p_w > p_v$ then
6:              $\mathbf{T_p} = \mathbf{T_p} - \{\theta_l, \theta_m, \theta_v, \theta_w\}$, $\mathbf{T_b} = \mathbf{T_b} \bigcup \theta_l, \theta_m, \theta_v, \theta_w$
7:              for all $\theta_z \in \mathbf{T_p}$ such that $S_{s_z} = S_i \wedge p_z > p_m$ do
8:                  $\mathbf{T_p} = \mathbf{T_p} - \theta_z$, $\mathbf{T_b} = \mathbf{T_b} \bigcup \theta_z$
9:              for all $\theta_y \in \mathbf{T_p}$ such that $S_{s_y} = S_j \wedge p_y > p_v$ do
10:                 $\mathbf{T_p} = \mathbf{T_p} - \theta_y$, $\mathbf{T_b} = \mathbf{T_b} \bigcup \theta_y$
```

Once the transitions of the FSM are partitioned, the behavior of the set $\mathbf{T_p}$ is implemented by a set of tasks generated according to the partitioned model, while the behavior of the set of transitions in $\mathbf{T_b}$ is implemented by a single task defined according to the baseline model and executing at the highest priority level, with deadline equal to its period (the *gcd* of all the events triggering the transitions in $\mathbf{T_b}$). The task generated according to the baseline model has an inhibit signal towards all the other tasks. Once again, all tasks share the variable encoding the FSM state. This implementation preserves the FSM untimed behavior because it preserves the priority of the transitions, executes only one transition (with highest priority) for each event and, with the deadline assignment rule specified for the two above models, guarantees that the state and output values are updated before the next event is processed to prevent any preemption while accessing the state variable.

Deferred output update model

The last model is a *deferred output update* model. In the FSM, each transition requires to compute the *output update* with the associated *actions*, followed by the *state update* function. This multi-task implementation *separates the two functions and maps them into different tasks*. The state update part is implemented using the baseline model, in a periodic task with highest priority, executed at the *gcd* of the events periods. The output update and action part (typically more computationally intensive) is realized using a partitioned model.

In addition, a function is executed together with the state update part to select the action and output update that need to be performed for the transition. The code in this task provides two more signals to the successor tasks: a signal cs indicating the current state (before the state update), and a indicating which action should be executed. The program code implementing the action and the output update part can then be implemented in tasks executed at the rates of the corresponding trigger events.

The state update function executes before the tasks handling the actions and output functions. This might seem to be conflict with the FSM semantics, as the output update function must execute before the state update because it needs to read the current state variable. However, it is sufficient to store and communicate the current state value in the signal cs, so that the execution order between the output generation and the state update functions can be reversed.

The tasks implementing the output update and action part will be assigned with deadlines equal to their periods, given that the need to update the state before the next event arrives is already satisfied by the gcd of the state update task.

IV. Examples

Consider a sample FSM with two trigger events, as in Figure 3, with its outputs connected to two follower blocks F_2 and F_3. The execution time (in bold, below the action) and priority (near the origin) of each transition are also denoted in the figure. o_1 is the input to F_2 executed with a period of $2ms$, and o_2 is the input to F_3 executed with a period of $5ms$.

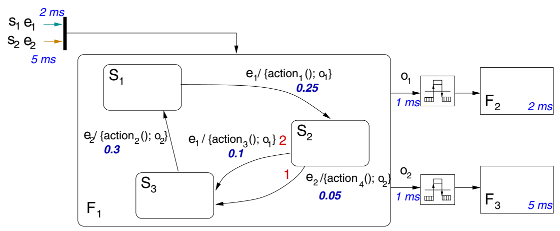


Fig. 3. An example of FSM with two trigger events

Figure 4 shows another example in which the FSM has an additional transition from S_1 to S_3 compared to Figure 3.

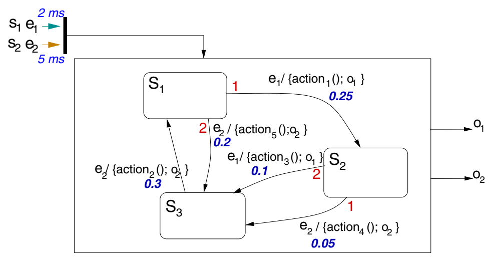


Fig. 4. Another example of FSM with two trigger events

Baseline implementation

If the FSM block in either Figure 3 or Figure 4 is implemented with a single task, its period is $1ms$ and two RT blocks, one for each output link, are required to guarantee flow preservation.

To show the possible disadvantage with the real-time feasibility of this implementation, suppose there is another task τ_3 with period of $2ms$ and worst case execution time (WCET) $1.65ms$ in addition to the FSM of Figure 4. The baseline implementation is unschedulable. The total execution time request from action$_3$ triggered by e_1 at time 4 and action$_3$ triggered by e_2 at 5 is $0.4ms$. This makes the instance of τ_3 activated at 4 miss its deadline at time 6. However, as we can see later, its mixed-partitioned implementation is schedulable.

Partitioned implementation

For the example FSM of Figure 3, all transitions associated with e_2 have higher priority than the ones associated with e_1. The corresponding two task implementation according to the partitioned model is shown in Figure 5. Task τ_1, on the top of the figure, implements the transitions and the associated actions activated by e_1. Its activation period is $2ms$. Task τ_2, on the bottom of the figure, executes with higher priority and implements the transitions and the associated actions activated by e_2. Its period is $5ms$. When e_1 and e_2 are associated with simultaneously enabled transitions, as in state S_2, the higher priority transition is taken and the other is disabled by the inhibition signal from τ_2 to τ_1 (the dashed line in Figure 5).

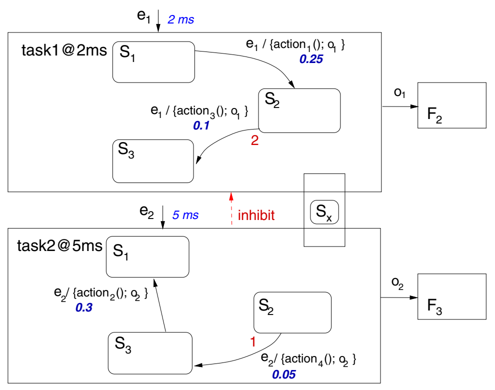


Fig. 5. The partitioned implementation of the example in Figure 3

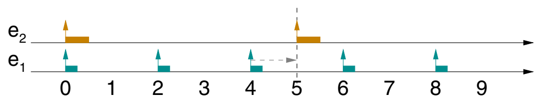


Fig. 6. The deadline of the multi-task implementation in Figure 5

As shown in Figure 6, the third instance of τ_1 in the hyperperiod has a deadline equal to $1ms$ because the reaction must update outputs and state to be used by the second instance of τ_2, activated $1ms$ later with a higher priority. All other instances of τ_1 have a deadline equal to the period ($2ms$) of the activation event e_1. The deadline of task τ_2 is always $5ms$ as it has a higher priority than τ_1. Compared to the baseline implementation of the same FSM, this multi-task implementation imposes a looser deadline than the single-task implementation (always with deadline of $1ms$) on most of the task instances, thus improving the feasibility.

Finally, as in the figure, both output links are now between tasks with the same rate, thus requiring no additional buffer.

Mixed-partitioned implementation

In the example of Figure 4, among the transitions from S_1, the one activated by e_1 has a higher priority than the one associated with e_2. For the state S_2, it is the opposite. Thus the partitioned implementation is not applicable to this example.

In the mixed-partitioned implementation (Figure 7), the transitions out of S_1 and S_2 are implemented by τ_1, and the remaining transition from S_3 is implemented by τ_2. τ_1 can be triggered by both e_1 and e_2 and runs at the gcd ($1ms$) of their periods, while τ_2 has the same period as e_2 at $5ms$.

We showed that the baseline model is unschedulable for the system with another task τ_3 (period $2ms$, WCET $1.65ms$) in addition to the FSM in Figure 4. In the mixed-partitioned implementation in Figure 7, we can assign τ_3 a priority higher than τ_2 but lower than τ_1. τ_3 is schedulable, as the maximum execution request from τ_1 within $2ms$ is no larger than $0.3ms$. The schedulability of τ_2 can also be validated by observing

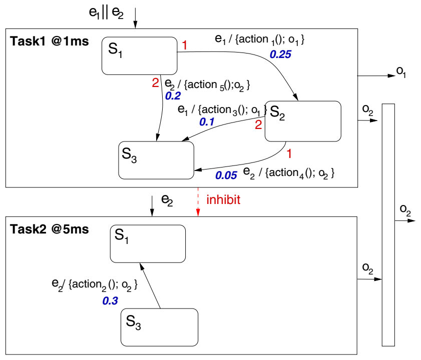
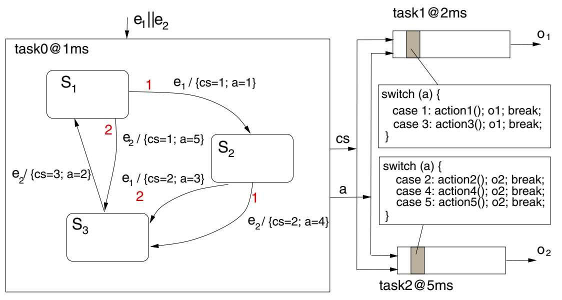


Fig. 7. The mixed-partitioned implementation of the example in Figure 4

that the maximum execution request from τ_1 and τ_3 in $5ms$ is no larger than $4.3ms$. This example highlights the case that a multi-task implementation gives the advantage in terms of real-time schedulability.

Deferred output update



```
switch (a) {
   case 1: action1(); o1; break;
   case 3: action3(); o1; break;
}
```

```
switch (a) {
   case 2: action2(); o2; break;
   case 4: action4(); o2; break;
   case 5: action5(); o2; break;
}
```

Fig. 8. The deferred output update implementation of the example in Fig. 4

Finally, Figure 8 illustrates the multi-task implementation of the example in Figure 4 according to the deferred output update model, which realizes the state update part of the FSM in task τ_0 on the left hand side, executed at $1ms$. Besides updating the state, τ_0 provides two more signals (cs, a) to the successor tasks τ_1 and τ_2, indicating the current state (before the state update) and the action τ_1 and τ_2 should execute. τ_1 is executed at the rate of e_1 ($2ms$) and updates output o_1, while τ_2 is executed at the rates of e_2 ($5ms$) and produces o_2.

V. Experiments

We generated random system configurations to explore the opportunities of improving memory efficiency. 1000 random systems are generated using the TGFF tool [5]. Each system contains 1 to 70 blocks with equal probability of being a Dataflow or Stateflow block. If the block is of type Dataflow, then a period of 10, 20, or 40 is randomly assigned to it. If it is of type Stateflow, it contains a maximum of 10 states with random transitions between them. To allow cyclic transitions in FSM blocks, we randomly reversed the direction of the edges generated by TGFF (limited to directed acyclic graphs). The period of the trigger event is also randomly chosen from 10, 20, or 40. Each transition updates 0, 1, or 2 outputs. The maximum degree of the blocks is 20, and the average is 3.

Among the 48585 links, if a single-task implementation is chosen, 27114 of them require the addition of RT blocks. However, if multi-task implementations are considered, there is more freedom to choose the periods for the writer and reader tasks of the links, thus possibly avoiding the RT blocks. Indeed, 2515 of the RT blocks can be avoided if a multi-task implementation is selected. This highlights the opportunity of improving memory efficiency by multi-task implementations.

The improvement of multi-task implementation on schedulability is measured on the estimated worst case processor utilization for the tasks implementing the randomly generated system configurations. As a simplistic analysis, we assume the WCET of a task is the WCET of all the transitions associated to it. The utilization of the single-task implementation is U_s. For multi-task implementations, we use the deferred output update model as it applies to all synchronous FSMs. The utilization of this multi-task implementation is denoted as U_m. For our 1000 random systems, the value of U_m/U_s varies between 34.3% and 100%, with an average of 59.8%.

VI. Conclusions and Future Work

In model-based design of embedded control systems consisting of blocks implementing finite state machines, commercial code generators define task models that are not efficient with respect to schedulability and memory usage. In this paper we provide solutions for the multitask implementation of finite state machine subsystems with improved schedulability. As a future work, we plan to find an analysis method that can predict with high accuracy the worst-case response time of a task implementing an SR FSM or a set of them. Furthermore, we plan to explore the opportunity to avoid the use of Rate Transition blocks in Simulink models consisting of both Dataflow and Stateflow blocks.

References

[1] A. Benveniste, B. Caillaud, L. Carloni, P. Caspi, and A. Sangiovanni-Vincentelli. Causality and scheduling constraints in heterogeneous reactive systems modeling. In *Formal Methods for Components and Objects*, *LNCS* 3188, Springer, 2004.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91, January 2003.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, November 1992.

[4] P. Caspi, N. Scaife, C. Sofronis, and S. Tripakis. Semantics-preserving multitask implementation of synchronous programs. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–40, January 2008.

[5] R. Dick, D. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. the 6th International Workshop on Hardware/Software Codesign*, 1998.

[6] M. Di Natale, G. Liangpeng, H. Zeng, and A. Sangiovanni-Vincentelli. Synthesis of multi-task implementations of Simulink models with minimum delays. *IEEE Transactions on Industrial Informatics*, 6(4):637–651, November 2010.

[7] dSPACE. *The dSPACE TargetLink Automatic Production Code Generator*. web page: http://www.dspaceinc.com.

[8] Mathworks. *The Mathworks Design Verifier User's Manuals*. web page: http://www.mathworks.com.

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

[10] M. Stigge, P. Ekberg, N. Guan, W. Yi. The Digraph Real-Time Task Model. In *Proceedings of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011.

[11] S. Tripakis, C. Sofronis, P. Caspi, and A. Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. on Embedded Computing Sys.*, 4(4):779–818, 2005.

[12] David Harel. Statecharts: A Visual Formalism for Complex Systems Science of Computer Programming, 1987.