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 tools 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 the 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-
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.

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 assume the use of RT blocks at every rate transition. However, in future work, we propose 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.

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 $(S, S_0, I, O, E, T)$, where $S = \{S_0, S_1, S_2, \ldots S_q\}$ is a set of states, $S_0 \in S$ is the initial state, $I = \{i_1, i_2, \ldots i_n\}$ and $O = \{o_1, o_2, \ldots o_p\}$ 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 $t_k = k_j \cdot t$, and are persistent between updates. $E$ 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
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 \( T = \{ \tau_1, \tau_2, \ldots, \tau_m \} \). If a transition \( \theta_j \) (and the corresponding action \( a_j \)) is mapped into \( \tau_i \), we write \( \mu(\theta_j) = \tau_i \). The task implementing \( \theta_j \) is also indicated as \( \tau(\theta_j) \). The period of \( \tau_j \) is indicated as \( \psi_j \) and its priority as \( \pi_j \).

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 \( \tau_1 \) implements all transitions. \( (\forall j, \mu(\theta_j) = \tau_1, \psi_1 = \text{gcd}_k \{ t_{e_j} \} \). 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 one 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 for each event (period). The tasks implementing the FSM are activated synchronously (with offset for each event (period)).

In this model, the FSM is implemented by one periodic task partitioned into two sets \( T_p \) and \( T_b \). The first set \( T_p \) 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 \( T_b \) 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 \( T_p \) and \( T_b \)

1: \( T_p = T \)
2: \( T_b = \{\} \)
3: for all \( s_i, s_j \in S \) do
4: for all \( t_i, t_m \in T_p \) with \( s_{x_{im}} = S_i \land p_i > p_m \) do
5: if \( \exists t_v, t_w \in T \) such that \( s_{x_v} = s_{x_w} = S_j \land e_{t_v} = e_{t_w} \land e_{t} = e_{t} \land e_{t_w} > p_v \land p_w > p_m \) then
6: \( T_p = T_p - \{t_i, t_m, t_v, t_w\}, T_b = T_b \cup \{t_i, t_m, t_v, t_w\} \)
7: for all \( t_i \in T_p \) such that \( s_{z_v} = S_i \land p_v > p_m \) do
8: \( T_p = T_p - \{t_i\}, T_b = T_b \cup \{t_i\} \)
9: for all \( t_i \in T_p \) such that \( s_{z_v} = S_j \land p_v > p_m \) do
10: \( T_p = T_p - \{t_i\}, T_b = T_b \cup \{t_i\} \)

Once the transitions of the FSM are partitioned, the behavior of the set \( T_p \) is implemented by a set of tasks generated according to the partitioned model, while the behavior of the set of transitions in \( T_b \) 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 \( T_b \)). 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 all 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 \( a \) 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 F2 and F3. The execution time (in bold, below the action) and priority (near the origin) of each transition are also denoted in the figure. o1 is the input to F2 executed with a period of 2 ms, and o2 is the input to F3 executed with a period of 5 ms.

![Figure 3. An example of FSM with two trigger events](image1)

Figure 4 shows another example in which the FSM has an additional transition from S1 to S3 compared to Figure 3.

![Figure 4. Another example of FSM with two trigger events](image2)

Baseline implementation

If the FSM block in either Figure 3 or Figure 4 is implemented with a single task, its period is 1 ms 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 τ3 with period of 2 ms and worst case execution time (WCET) 1.65 ms in addition to the FSM of Figure 4. The baseline implementation is unschedulable. The total execution time request from action3 triggered by e1 at time 4 and action3 triggered by e2 at 5 is 0.45 ms. This makes the instance of τ3 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 e2 have higher priority than the ones associated with e1. The corresponding two task implementation according to the partitioned model is shown in Figure 5. Task τ1, on the top of the figure, implements the transitions and the associated actions activated by e1. Its activation period is 2 ms. Task τ2, on the bottom of the figure, executes with higher priority and implements the transitions and the associated actions activated by e2. Its period is 5 ms. When e1 and e2 are associated with simultaneously enabled transitions, as in state S2, the higher priority transition is taken and the other is disabled by the inhibition signal from τ2 to τ1 (the dashed line in Figure 5).

![Figure 5. The partitioned implementation of the example in Figure 3](image3)

As shown in Figure 6, the third instance of τ1 in the hyperperiod has a deadline equal to 1 ms because the reaction must update outputs and state to be used by the second instance of τ2, activated 1 ms later with a higher priority. All other instances of τ1 have a deadline equal to the period (2 ms) of the activation event e1. The deadline of task τ2 is always 5 ms as it has a higher priority than τ1. 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 1 ms) 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 S1, the one activated by e1 has a higher priority than the one associated with e2. For the state S2, 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 S1 and S2 are implemented by τ1, and the remaining transition from S3 is implemented by τ2. τ1 can be triggered by both e1 and e2 and runs at the gcd (1 ms) of their periods, while τ2 has the same period as e2 at 5 ms.

We showed that the baseline model is unschedulable for the system with another task τ3 (period 2 ms, WCET 1.65 ms) in addition to the FSM in Figure 4. In the mixed-partitioned implementation in Figure 7, we can assign τ3 a priority higher than τ2 but lower than τ1. τ3 is schedulable, as the maximum execution request from τ1 within 2 ms is no larger than 0.3 ms. The schedulability of τ2 can also be validated by observing
that the maximum execution request from $\tau_1$ and $\tau_3$ 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**

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 $\tau_0$ on the left hand side, executed at 1ms. Besides updating the state, $\tau_0$ provides two more signals ($cs$, $a$) to the successor tasks $\tau_1$ and $\tau_2$, indicating the current state (before the state update) and the action $\tau_1$ and $\tau_2$ should execute. $\tau_1$ is executed at the rate of $e_1$ (2ms) and updates output o1, while $\tau_2$ is executed at the rates of $e_2$ (5ms) and produces o2.

**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 with 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 multi-task 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**


