Task Implementation and Schedulability Analysis of Synchronous Finite State Machines

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

Haibo Zeng
GM R&D Palo Alto, CA, haibo.zeng@gm.com

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. Previous research focused on the concurrent implementation of the dataflow part of SR models, including the optimization of the block-to-task mapping and the optimization of the communication buffers. When the system is composed of blocks implementing finite state machines, as in 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 and analysis of finite state machine subsystems. We define the constraints and efficient policies for the block to task implementation and we introduce the schedulability analysis of these systems.

1 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 or formalism, are being adopted to increase productivity and reduce the number of errors in the development of embedded software.

In the development of embedded controllers, the use of the Simulink formalism and modeling tool, together with associated code generation tools such as Real-Time Workshop (RTW) and Embedded Coder (EC) from MathWorks and TargetLink of dSPACE is becoming widespread. 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 description [6] for the purpose of formal verification of properties and/or provably correct code generation. Commercial products (e.g., Design Verifier) for the verification of properties of Simulink models and for the automatic generation of tests with guaranteed coverage are also available.

Traditional (worst case) schedulability analysis is based on a black-box model of tasks. With the advent of model-based design and automatic code generation tools, tasks are increasingly the result of automatic optimization techniques and knowledge about the models and the system reaction attributes is the starting point. This additional knowledge can be leveraged in several ways: to build an efficient task model according to some constraints or performance metrics of interest, or to build a more accurate model of the task for the purpose of timing analysis. In this work we focus on the very popular finite state machine (FSM) abstraction and we discuss schedulability model for synchronous FSMs.

A block implementing an FSM receives as input a set of signals and a set of events obtained from other signals. As a result of its reaction, the block updates a set of output signals. The events triggering the reaction of the block are obtained from periodic signals and can be assumed periodic themselves.

There is plenty of work on task modeling frameworks in the literature. We borrow the concept of *task graph* to illustrate the expressiveness of each model framework. A task in the model can be represented by a task graph where the vertices represents the subtasks and the edges are the possible flows of control. The task graphs for the existing task models are: a *single vertex* for classic periodic task model [8] and its generalization on the possible activations (sporadic tasks, or with explicit deadline); a *chain of vertices* for the multiframe model [9] and generalized multiframe task model [3]; *directed trees* for the recurring branching task model [2], and [1] which generalizes [2] by allowing for branches of different length; *directed acyclic graph* for the recurring real-time task model in [4]. Timed automata with tasks [10] is a generalization of all the above models, which allows the underlying task graph to be any structure. The code implementation of synchronous state machines has been discussed at length by Berry [12] Benveniste and Caspi [14]. 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 implementa-

tions previous works focused mostly on the implementation of the intertask communication [13], rather than the real-time analysis or the synthesis of an efficient task structure. On the commercial side, the problem of guaranteeing flow preservation in a multitask implementation of communicating blocks executing at different rates is solved by Simulink using Rate Transition (RT) blocks. A Rate Transition block buffers any transition between blocks at different rates. It behaves like a Zero-Order Hold block for fast to slow transitions, or a Unit Delay block plus a Hold block (Sample and Hold) for slow to fast transitions.

RT blocks for fast to slow transitions require an additional set of output variables for communication between the sender and the receiver and additional code for the output update function of the block. The memory overhead is equal to the size of the communication link. RT blocks for slow to fast transitions require additional sets of state and output variables and 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 blocks results in an additional functional delay equal to the period of the slower block.

Our contributions The code generation methods of commercial tools like EC/RTW and SCADE implementations assume a single periodic task implementation for each FSM. However, a multi-task implementation can provide more flexibility and efficiency in terms of schedulability. In the case of Stateflow blocks communicating with other blocks in a multirate model, a multitask implementation can avoid the addition of Rate Transition (RT) blocks between tasks with different periods, saving on the associated memory overhead. In this work, we reason about the possible multitask implementations for FSMs and their benefits on real-time schedulability and memory usage.

With the observation that the task graph of an FSM can be naturally cyclic, the currently available task models either lack enough expressive power to capture the FSM behavior, or can be exponentially difficult to calculate the demand and request bound functions for schedulability analysis. This calls into the question of practical viability of an exact schedulability analysis for synchronous FSM task models, especially for large size problems. In this work, we look at the approximation algorithms which are fast but provide tight bound on the demand and request bound functions.

2 Synchronous FSM abstract model

A synchronous model consists of a graph of communicating Mealy finite state machines. Each FSM block can be characterized by a set of input signals $\{i_1, i_2, \dots i_n\}$, a set of trigger events $\{e_1, e_2, \dots e_m\}$ and a set of output signals $\{o_1, o_2, \dots o_p\}$. The internal behavior follows the semantics and notation of extended (hierarchical and concurrent) Statecharts. The FSM is defined by a tuple $\{\mathbf{S}, S_0, \mathbf{I}, \mathbf{O}, \mathbf{E}, \mathbf{T}\}$, where $\mathbf{S} = \{S_1, S_2, \dots S_q\}$ is a set of *states*, $S_0 \in \mathbf{S}$ is the initial state, $\mathbf{I} = \{i_1, i_2, \dots i_n\}$ and $\mathbf{O} = \{o_1, o_2, \dots o_p\}$ 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 is defined as $k \cdot t_j$, where t_j is the signal's period. $\mathbf{E}$ are the activation (or trigger) events. Each event e_j is bound to occur only at time instants belonging to a periodic time base, with period t_j. However, at each time $k \cdot t_j$ the event may or may not be present (alternatively, the machine will stutter). The periods of all signals and events are multiple of a *base period* t_0, with phase 0. $\mathbf{T}$ is the set of transition rules, where each $\theta_j \in \mathbf{T}$ 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 an event, g_j is a guard condition (an expression of the input and output signals and internal values values), a_j is an *action* and p_j is the transition priority. Priorities associated to transitions are used to discriminate which transition should be performed when two or more events are active and two or more corresponding transitions can be taken out of a state. The transition with higher priority (smaller p) is selected. Figure 1 shows an example of the graphical notation that is used to describe states, transitions and the events, guards and actions associated with each state. Following the original Statecharts specification, from which it is derived, the actual Stateflow semantics also allow *concurrent states*, *superstates*, *entry* actions, *exit* actions, and *while* actions, *join transitions* and other constructs. However, while these extensions simplify the definition and the structure of FSMs, they are semantically equivalent, and can be translated, to the definitions for standard (flat) FSMs as described earlier in the paragraph.

Synchronous FSM adds constraints on the events and the reactions of the FSM. All events occur with periods as multiples of the base period and with the same phase, therefore events arriving at the same time are quite common. Also, the reaction of the FSM 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.

3 Task Implementation of the FSM

The RTW/EC code generator implicitly assumes a single-task implementation. According to this model, the task period is assigned as the greatest common divisor of the period of the triggering events. For example, the single task implementation of FSM of Figure 1 has a period of 1ms. However, this is not the only option for the implementation. As shown in Section 3.2, several possible multi-task

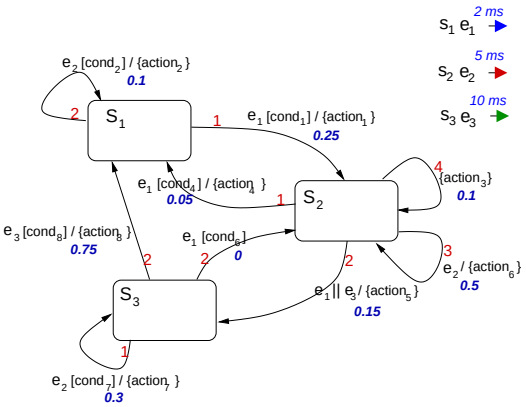

Figure 1. An example of FSM description

implementations exist, which may give the advantage in the following two aspects: 1) real-time schedulability; 2) memory usage. Multi-task implementations give at least as much design freedom as a single-task implementation on block to task mapping and priority assignment. In addition, multi-task implementations possibly impose looser deadline requirements. The possible advantage on the memory side relates to the possibility of avaoding rate transition blocks altogether.

3.1 Single-task Implementation

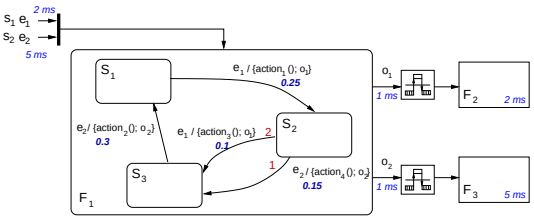

Figure 2. AnFSM with two trigger events

We first look at an example with two trigger events, as in Figure 2. Suppose the FSM connects to two other blocks, where o_1 is the input to F_1 with a period 2ms, and o_2 the input to F_2 with a period 5ms. The Simulink code generation tools will adapt a Rate Transition (RT) block on both outgoing links. For the example in Figure 2, if the FSM is implemented with a single task, its period is assigned as 1ms and thus two RT blocks, one for each output link, are required to guarantee the semantics preservation of the generated code.

3.2 Multi-task Implementation

A multi-task implementation can possibly save some of this memory overhead. For the example in Figure 2, there is only one state S_2 with multiple out-going transitions, and the transition activated with event e_2 has a higher priority than the one with e_1. Based on this fact, a two-task implementation, one task for the transitions associated with each event e_1 and e_2, is shown in Figure 3. Task τ_1, on the top of the figure, implements the transitions and the associated actions activated by event e_1, thus its period is the same as e_1

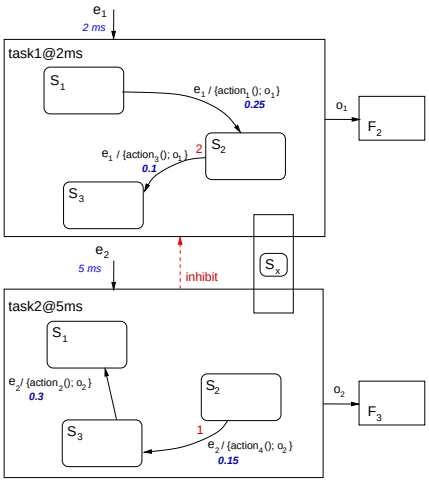

Figure 3. The multi-task implementation of the example in Figure 2

(2ms). Task τ_2, on the bottom of the figure, implements the transitions and the associated actions activated by event e_2, thus its period can be regarded as 5ms. When both e_1 and e_2 appear, then the transition associated with the higher priority should be selected and the other transition should be disabled. This can be implemented by adding a signal from τ_2 to τ_1 (denoted by the dashed line in Figure 3): whenever a transition from τ_2 is enabled, this signal is on and τ_1 is disabled. This implementation is possible because transitions activated by e_2 (implemented in τ_2) always have priority higher than those associated with e_1 (implemented in τ_1).

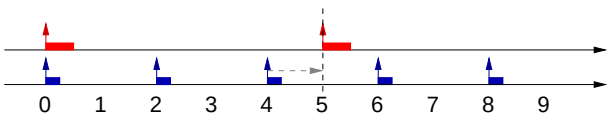

Figure 4. The deadline of the multi-task implementation in Figure 3

As in Figure 4, the third instance of τ_1 in the hyperperiod, has a deadline equal to 1ms because the second instance of τ_2 is activated 1ms later with a higher priority than τ_1. All the other instances of τ_1 have a deadline equal to the period (2ms). The deadline of task τ_2 is always 5ms as it has a higher priority than τ_1. The state variable (denoted as S_x in the figure) is shared between task τ_1 and τ_2. This sharing, however, does not require any protection mechanism to ensure data consistency because preemption between τ_1 and τ_2 is not possible as long as task deadlines are met.

Clearly, not all FSMs are defined in such a way that all transitions from the same source state can be ordered by priority according to the triggering event. An example is the FSM of Figure 2 with an additional transition from S_1 to S_3, activated by e_1 with priority higher priority than the one associated with e_2.

In this case, one possible implementation consists of

mapping the transitions triggered by events for which there is no strict priority order into one task, executed at the greatest common divisor of the events periods. The other transitions can be implemented by one task for each event, executed at the corresponding rate. For our example, the transitions out of S_1 and S_2 are implemented in task τ_1, and the remaining transition from S_3, is implemented in τ_2. τ_1 can be triggered by both e_1 and e_2, thus runs at the gcd period of 1ms, while τ_2 has the same period as e_2 at 5ms.

4 Schedulability Analysis of the FSM

When modeling the FSM for schedulability, besides the structure of the machine and the rate information about its trigger events, it is also necessary to associate a worst-case execution time γ_j to each action a_j. For schedulability analysis of these task models, two concepts, the Demand Bound Function (DBF) and Request Bound Function (RBF) [5], are very useful. The request (demand) bound function of task τ_i in the interval of length t, denoted as $rbf_i(t)$ ($dbf_i(t)$), is defined as the maximum amount of cumulative execution requirement by jobs of τ_i that have activation times (both activation times and deadlines) within any time interval of length t. For tasks with static priority and preemptive scheduling, a sufficient schedulability condition is

$$\forall t, \exists t' \leq t \text{ s.t. } t' - \sum_{j \in hp(i)} rbf_j(t') \geq dbf_i(t) \qquad (1)$$

where $hp(i)$ is the set of tasks with priority higher than τ_i.

According to the classic periodic task model [8], the FSM of the example of Figure 1 has a period of 1ms and a worst-case execution time of 0.75ms. This estimate, however is very pessimistic and can be easily improved upon. In the example, the event sequence in the hyperperiod are $\{e_1||e_2||e_3, e_1, e_1, e_2, e_1, e_1\}$, and the RBF is obtained by an exhaustive search of the reachable states reacting to this event sequence. The highest load within one hyperperiod (10ms) is 1.9ms when starting from S_3, as in Figure 5.

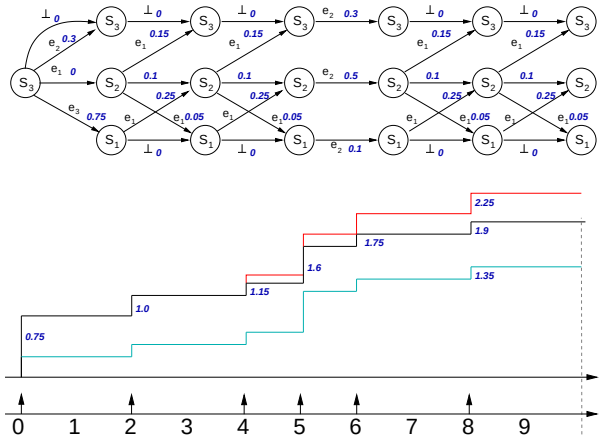

Figure 5. A better upper bound on the demand bound function (starting from S_3)

5 Conclusions and Future Work

When the system is composed of blocks implementing finite state machines, as in modern modeling tools like Simulink and SCADE, the previous techniques can be applied with significant pessimism. 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 block to task implementation and we introduce the schedulability analysis of these systems. We plan to formulate it as an optimization problem to find out the best task implementation for such systems with respect to memory usage or functional delays.

References

[1] Madhukar Anand. *Conditional Models for Compositional Design of Real-time Embedded Systems*. PhD thesis, University of Pennsylvania, Janurary 2008.

[2] S. Baruah. Feasibility analysis of recurring branching tasks. *Real-Time Syst., Euromicro Workshop on*, 1998.

[3] S. Baruah, et al. Generalized multiframe tasks. *Real-Time Syst.*, 17(1):5–22, 1999.

[4] S. Baruah. Dynamic- and static-priority scheduling of recurring real-time tasks. *Real-Time Syst.*, 24(1):93–128, 2003.

[5] S. Baruah, et al. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Syst.*, 2:301–324, October 1990.

[6] A. Benveniste, et al. The synchronous languages 12 years later. *Proc. of the IEEE*, 91, January 2003.

[7] S. Chakraborty, et al. On the complexity of scheduling conditional real-time code. In *Proc. of Workshop on Algorithms and Data Structures*, pages 38–49, 2001.

[8] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[9] A. K. Mok and D. Chen. A multiframe model for real-time tasks. In *RTSS '96*, 22–29, 1996.

[10] C. Norström, et al. Timed automata as task models for event-driven systems. In *RTCSA '99*, 182–189, 1999.

[11] S. Tripakis, et al. Translating discrete-time simulink to lustre. *Trans. on Embedded Computing Sys.*, 4(4):779–818, 2005.

[12] G. Berry, G. Gonthier The Esterel Synchronous Programming Language: Design, Semantics, Implementation Science of Computer Programming vol. 19, n2, pp 87-152, 1992.

[13] P. Caspi, N. Scaife, C. Sofronis, S. Tripakis Semantics-preserving multitask implementation of synchronous programs ACM Transactions on Embedded Computing Systems, Vol. 7 (2), February 2008

[14] A. Benveniste, B. Caillaud, L. P. Carloni, P. Caspi, A. L. Sangiovanni-Vincentelli Causality and Scheduling Constraints in Heterogeneous Reactive Systems Modeling. FMCO 2003: 1-16