Schedulability Analysis of Periodic Tasks Implementing Synchronous Finite State Machines

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

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

***Abstract*—Model-based design of embedded systems using Synchronous Reactive (SR) models is among the best practices for software development in the automotive and aeronautics industry. The correct implementation of an SR model must guarantee the synchronous assumption, that is, all the system reactions complete before the next event. This assumption can be verified using schedulability analysis, but the analysis can be quite challenging when the system also consists of blocks implementing finite state machines, as in modern modeling tools like Simulink and SCADE. In this paper, we discuss the schedulability analysis of such systems, including the applicability of traditional task analysis methods and an algorithmic solution to compute the exact *demand and request bound functions*. In addition, we define conditions for computing these functions using a periodic recurrent term, even when there is no cyclic recurrent behavior in the model.**

I. Introduction

The development of complex embedded systems is subject to tight cost and performance constraints. Automatic code generation tools, which produce a software implementation of an application model defined according to a high-level visual language, are being adopted to increase productivity and reduce design errors. Among them, the use of the Simulink language and modeling tool, together with the associated code generators such as Real-Time Workshop (RTW), Embedded Coder (EC) from MathWorks [1] and TargetLink of dSPACE [2], is becoming widespread.

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 output signals as a function of the current state and the inputs. For the purpose of this work, we are interested in the subset of Simulink/Stateflow models that allows the automatic generation of a code implementation, that is, blocks with discrete time input and output signals in a model with a fixed-step solver.

Simulink blocks are of two types. *Dataflow* blocks are almost invariably executed at their periods (integer multiples of the system-wide *base period*). Other blocks are (Mealy type) *Extended Finite State Machines*, or FSMs, called *Stateflow* blocks. In Stateflow blocks, each event (whether it causes a state transition or not) may also trigger the execution of a set of *actions* (functions defined by designers).

A Simulink Stateflow block 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 that trigger the reaction of the block are obtained from periodic signals and are therefore assumed to be periodic*. If no trigger event is specified, the Stateflow block reacts according to the model base rate.

When implementing a Simulink model on a target execution platform, the problem is to provide a feasible code implementation (for example, with respect to time and memory constraints) that preserves the logical-time execution semantics (the rate and order of execution of the blocks and the communication flows). In a single-task implementation, all blocks are executed by a task running at the system base period according to a global order compliant with the partial order execution constraints. Feasibility requires that the longest reaction in the system completes before the end of the base period. In multi-threaded implementations, all blocks with the same rate are executed by the same task and tasks are scheduled by priority.

The code generation of the Embedded Coder/Real-Time Workshop [1] tools assumes a single periodic task implementation for each Stateflow block code. The period of the task is the greatest common divisor of the periods of the trigger signals. Each time the task is activated, it checks for any active trigger and, if there is any, it processes them. This work assumes the existence of a semantics-preserving implementation of a set of synchronous FSMs as a set of periodic tasks scheduled by priority [10].

A. Relationship with existing task models

Most of traditional real-time analysis originates from the study of concurrent programming languages rather than software models. The analysis units are tasks: units of sequential code implementing the system actions and executing concurrently (scheduled by an operating system).

A classification of task models can be attempted based on the concept of *task graph*. A task in the model is represented by a graph, where the vertices represent different kinds of jobs, and the edges are the possible flows of control. Each

vertex (or type of job) is characterized by the worst-case execution time and relative deadline. Each edge is labeled with the minimum separation time between the release of the two vertices it connects.

A *single vertex* task graph corresponds to the simplest model of independent tasks activated by periodic or sporadic events. The multiframe model [16] and generalized multiframe (GMF) task model [6] assume that worst case execution times are not constant for each job instance, but defined according to a cyclic pattern. The corresponding task graph is therefore a *chain of vertices*. The recurring branching task model [7] allows selection points to determine which task behavior should occur for a given instance, in statements such as "if-then-else" and "case", thus modeling conditional branches and optional (OR-type) execution. The corresponding task graph representation is a *directed tree*. The recurring real-time task model (RRT) [8] allows the task graph to be any *directed acyclic graph* (DAG). All the above models satisfy the property of *cyclic recurrent behavior*:

- **Recurrent**: the graph has a unique source vertex. The completion of a sink vertex job automatically releases the source vertex job. This execution pattern may be implicit, or can be modeled by explicitly adding back edges from sink vertices to the unique source vertex.
- **Cyclic**: a parameter defines the minimum time interval (for sporadic executions, or the period, in case of periodic executions) that must elapse between two consecutive releases of the source vertex job.

In addition, these models are typically characterized by the **frame separation property**: job deadlines are no larger than the inter-arrival time of any outgoing edge.

The concepts of *request bound function* (or *rbf*) and *demand bound function* (or *dbf*) have been introduced in [8] for the analysis of task graphs. For a task graph, the maximum cumulative execution times by jobs that have their activation time within any time interval of length t is defined as its request bound function $rbf(t)$. The property of cyclic recurrent behavior is leveraged to find the periodicity of these functions, such that the asymptotic complexity is independent from the time interval length t.

The non-cyclic generalized multiframe model [18] removes the cyclic parameter in the activation pattern of the frame jobs. More specifically, it is possible to activate any job as long as the minimum separation time with respect to its predecessor is satisfied. The task graph model proposed in [4] is a generalization of the recurring branching model [7] since it allows for branches of different length (anisochronicity). With the exception of the (possibly implicit) back edges from the sink vertices to the source vertex, the task graph is still restricted to be a *directed tree*. The non-cyclic recurring real-time task model [9] is a generalization of both the recurring real-time [8] and the non-cyclic GMF [18] models, since it removes the restriction of requiring an execution period parameter applied to the root job. These three models [4] [18] [9] relax the constraint of cyclic execution of the graph, but still assume a model with a single root vertex (previously defined as **recurrent**).

The digraph model [19] removes the restriction of having a single root job by allowing arbitrary directed graphs to represent the release structure of jobs, which significantly increases the expressiveness. Arbitrary cycles are allowed in the digraph model. Another generalization is to relax the *frame separation property* to allow arbitrary job deadlines. The extended digraph model [20] adds global minimum inter-release constraints between any two vertices, including those that have no connecting edge (or functional dependency). The schedulability analysis is tractable (pseudo-polynomial time) for bounded-utilization systems. However, as in all the previously mentioned models, the edges are still labeled with a minimum inter-arrival time, which is not (as shown later) a natural way to capture the periodicity of trigger events for multi-rate finite state machines.

Timed automata with tasks are a generalization of all the above models, which allow for complex dependencies between job release times and task synchronization. However, schedulability analysis is shown to be very expensive and even undecidable in certain variants of the model [11].

A discussion of the task models and their generalization relationship can be found in e.g. [9] [19]. As the (extended) digraph task model is the most general one whose schedulability analysis is still tractable, we examine the possibility to express synchronous multi-rate finite state machines using the digraph model in Section II, despite its limitations.

Our contribution: In this work, we discuss the conditions and an algorithmic solution for the exact computation of the *demand bound function* and *request bound function*, which are the inputs to the schedulability analysis of synchronous finite state machine models. In particular, we define the conditions for the periodicity of these functions, even when there is no cyclic recurrent behavior in the model. As synchronous FSMs are triggered by synchronized periodic events, unlike the analysis in [19] where tasks are assumed to have arbitrary offsets, the schedulability analysis must consider periodic tasks with synchronized offsets.

II. Synchronous FSM abstract model

The synchronous model studied in this paper consists of a graph of communicating Mealy Finite State Machines (FSMs). Each FSM block can be characterized by a set of input signals, a set of trigger events, and a set of output signals. The internal behavior of a Simulink Stateflow block follows the semantics and notation of extended (hierarchical and concurrent) state machines. In this paper, we consider for simplicity the case of simple (flat) FSMs, without concurrency and hierarchy, as well as other Stateflow extensions and notational conveniences.

An FSM is defined by a tuple $\mathcal{F} = \{\mathbf{S}, s_\alpha, \mathbf{I}, \mathbf{O}, \mathbf{E}, \mathbf{\Theta}, \mathbf{V}\}$, where $\mathbf{S} = \{s_1, s_2, \ldots s_{|\mathbf{S}|}\}$ is a set

of *states*, $s_\alpha \in \mathbf{S}$ is the initial state, $\mathbf{I} = \{i_1, i_2, \dots i_{|\mathbf{I}|}\}$ and $\mathbf{O} = \{o_1, o_2, \dots o_{|\mathbf{O}|}\}$ are the *input* and *output* signals and $\mathbf{V}$ is a set of internal variables. Each i_j (o_j) is also denoted as α_j and is a function defined on a discrete time domain with values in a given range. The discrete time domain of each signal α_j matches the system *base period* T_b. Signal values only change at multiples of its period, defined as $T_{\alpha_j} = k_{\alpha_j} \cdot T_b$, and are persistent between updates.

$\mathbf{E}$ is the set of activation or trigger events. Each event e_j occurs at (rising or falling) edges of a signal, and therefore occurs only at time instants belonging to a time base with period T_{e_j}, an integer multiple of T_b. At each time $k \cdot T_{e_j}$ the event may be present (if there is an edge on the signal) or absent (otherwise). T_{e_j} is the period of the event.

To define the behavior at each instant kT_b, even when no event is present, the notion of *stuttering behavior* is conventionally added ([22] is a textbook describing stuttering as well as some of the composition rules for transforming hierarchical state machines to flat FSMs). Formally, an *absent* event, denoted as $\perp$, is added to the set of input events and assumed as present at all multiples of the base period when no event e_j is set. The reaction of the Stateflow chart to the absent event is the following: the state does not change, neither do the output signal values.

$\mathbf{\Theta}$ is the set of transitions. Each transition $\theta_j \in \mathbf{\Theta}$ consists of a tuple $\theta_j = \{\text{src}(\theta_j), \text{snk}(\theta_j), e(\theta_j), g_j, a_j, p_j\}$, where a_j is the *action*, $\text{src}(\theta_j)$ is the source state, $\text{snk}(\theta_j)$ is the sink state, $e(\theta_j) \in \mathbf{E}$ is the trigger event, g_j is the guard condition (an expression of the input and output signals and internal variable values), and p_j is the transition priority (the lower the number, the higher the priority), which determines the order by which the conditions for the transitions to be active (according to event presence and guard) are evaluated. Priorities determine which transition should be taken when two or more transitions can be taken out of a state at the same time instant. The event triggering a specific action a_j may also be labeled as $e(a_j)$ and its period as $T_{e(a_j)}$.

Given that events are conditionally generated based on signals, guards have no effect on worst case timing analysis. Because of guards there may be cases in which, despite an event is set, the transition is not taken, but events are already assumed as possibly absent on some of their periodic instances. In our model, guards are included for a better match to real-world FSM models (Stateflow).

For timing analysis, we assume each action a_j is characterized by its worst case execution time (WCET) C_{a_j}. The hyperperiod of an FSM $H_\mathcal{F}$ is the least common multiple of all the periods of its events. The hyperperiod of the FSM is also the period of the corresponding Stateflow block. The system hyperperiod H is defined as the least common multiple of all the (hyper)periods of the blocks (of type Dataflow or Stateflow).

Figure 1 shows an example of the notation used to describe the states and transitions, along with the event, guard and action associated to each transition. The execution time (in grey) and priority (red/dark, near the source state) of each transition are also denoted in the figure.

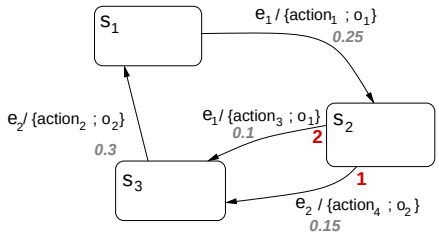

Figure 1. An example of FSM behavior description. The guard conditions are empty thus omitted.

Please note that priorities are associated to transitions to make FSM behaviors deterministic that would otherwise be nondeterministic. For example, in the FSM of Figure 1, if events e_1 and e_2 have periods $2ms$ and $5ms$, they will occur *simultaneously* every $10ms$. If the system is in state s_2, the priorities associated with the outgoing transitions indicate that the transition with action a_4 (priority 1) will be taken, not the transition with action a_3 (priority 2).

Following the original Statecharts specification from which they are derived, the actual Stateflow semantics also allow *concurrent states*, *superstates*, *entry* actions, *exit* actions, *while* actions, *join transitions*, and others. Also, in Stateflow (as in most extended FSM formalisms), transitions can be triggered by a logical combination of events. An FSM with hierarchical superstates with AND/OR composition can be transformed in many cases into an equivalent flat FSM with states obtained from the set product of the states of the composed machines (examples can be found in the seminal paper [23], and the procedure is defined in [22] for parallel and series compositions). Of course, the resulting number of states can be very high and a general type of transformation may not exist when all the other Stateflow semantic extensions are considered. For simplicity, in this paper we assume standard (flat) FSMs, in which each transition is associated with a single event.

In synchronous FSMs all events occur with periods that are multiples 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*, i.e. it satisfies the synchronous assumption, where the reaction of a network of (possibly FSM) blocks completes before the next event is processed. We use $\mathbf{T}_\mathcal{F}$ to denote the *ordered* set of time instants that are integer multiples of event periods of FSM $\mathcal{F}$

$$\forall t \in \mathbf{T}_\mathcal{F}, \exists e_i \in \mathbf{E} \text{ and } k \in \mathbb{N} \text{ such that } t = k \times T_{e_j},$$

where $\mathbb{N}$ is the set of non-negative integers. By the semantics of Stateflow, the release times and absolute deadlines of $\mathcal{F}$ are members of the set $\mathbf{T}_\mathcal{F}$.

When an FSM is translated into code, for each possible state and input signal, the task implementing (a subset

of) the reactions must define the *state update* and *output update* functions with the corresponding *actions* (the two are sometimes merged). *In this paper, we assume a single-task implementation for each FSM, and the terms FSM and task are used interchangeably*. Multi-task implementations are possible ([10]) and, while we believe our analysis can be easily extended to other models, the formal proof is beyond the scope of this work and left to future extensions.

III. Digraph task model for synchronous FSMs

In synchronous FSMs, each action can be characterized by its WCET. The relative deadlines and the minimum inter-arrival time separating the actions are not uniquely determined by the action, but also depend on the pattern of trigger events in the hyperperiod.

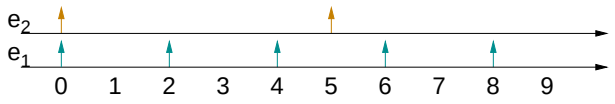

Figure 2. The pattern of the trigger events in the first hyperperiod

The FSM in Figure 1 with four transitions will be used as an example. In the corresponding digraph model, we first consider each action as a different type of job, i.e. one vertex for each action. Figure 2 shows the event stream pattern in the first hyperperiod. The minimum inter-arrival time between e_1 and e_2 is $1ms$, the greatest common divisor of the periods of the two events. Thus, the edges (a_1, a_4), (a_2, a_1), and (a_3, a_2) are labeled with $1ms$. a_1 and a_3 are both triggered by e_1, thus the edge (a_1, a_3) is labeled with $2ms$, the period of e_1. Similarly, (a_4, a_2) is labeled with $5ms$, the period of e_2. Because of the synchronous assumption, the deadline of an action is defined as the minimum distance to the next trigger event, which equals to the minimum label among all the outgoing edges (thus satisfying the frame separation property). For example, vertex a_2 is labeled with the <WCET, deadline> pair <0.3, 1>. The result digraph task model is shown in Figure 3.

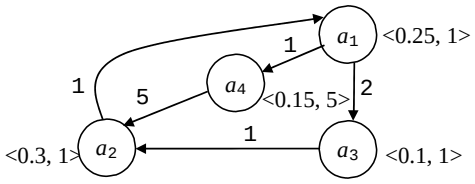

Figure 3. The digraph task model for the FSM in Figure 1

However, the task model in Figure 3 is pessimistic, as the periodic pattern of event streams is not adequately captured. For example, the model allows the activation of a_2, a_1, and a_4 in a time interval of $2ms$, which implies that the event sequence e_2, e_1, e_2 occurs in a $2ms$ interval. This is impossible with reference to the event pattern of Figure 2.

A more accurate task model is obtained by identifying each action occurrence according to the time when its trigger events happen in the hyperperiod. For each action a_i, we generate an infinite sequence of vertices in the digraph model, each vertex $A_{i,t_{i,k}}$ corresponds to a_i triggered by an event at time $t_{i,k}$. As its trigger event $e(a_i)$ is periodic, $t_{i,k} = k \times T_{e(a_i)}$, $\forall k \in \mathbb{N}$. In our example, there will be a set of vertices $A_{2,t_{2,k}}$, $t_{2,k} = 0, 5, 10, ...$ representing the possible executions of a_2 at times $t_{2,k} = 5 \times k$. The next action following a_2 is a_1. Since events may also be inactive (the machine may stutter), following $A_{2,0}$ (action a_2 executed at time 0), action a_1 could then be executed at 2, 4, 6, and so on. Thus, in the task graph there should be an edge from $A_{2,0}$ to $A_{1,j}$ for $j = 2, 4, 6, ...$ Generally, for each $A_{i,t_{i,k}}$, the action a_{i+1} possibly following a_i can be triggered by the event $e(a_{i+1})$ at time $t_{i+1} = (k + \lfloor \frac{t_i}{T_{e(a_{i+1})}} \rfloor) \times T_{e(a_{i+1})}$, for all positive integers k ($\forall k \in \mathbb{N}^+$).

The complete digraph model for our example is shown in Figure 4, where for simplicity only edges from vertices at time 0 are shown. The minimum inter-arrival time label on each edge $(A_{i,k} - A_{j,l})$ is omitted, but can be easily computed as $(l - k)$. Since there is an infinite number of vertices and edges in the figure, schedulability analysis requires the reduction of this digraph.

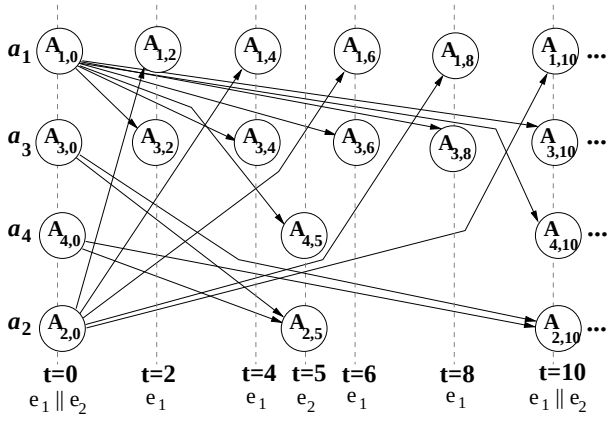

Figure 4. A more accurate digraph task model for the FSM in Figure 1. To be clear, only edges from vertices at time 0 are shown

The digraph model can be simplified by removing the edges that are not critical to the schedulability analysis, and by folding vertices by exploiting the periodic pattern of the event arrivals in the hyperperiod. We introduce the concept of *tight* and *loose* edges, and restate the definitions of *action sequence* and *request (demand) critical actions*, as introduced in [8] (the original definition is event sequence).

Definition 1: Edge $(A_{i,t_{i,k}}, A_{j,t_{j,q}})$ in the digraph model is *tight* if there is no other edge $(A_{i,t_{i,k}}, A_{j,t_{j,p}})$ in the model such that $t_{i,k} < t_{j,p} < t_{j,q}$ (that is, $p < q$). In other words, $t_{j,q}$ is the earliest time a_j can be triggered following a_i at time $t_{i,k}$. All other edges are defined as *loose*.

Definition 2: An *action sequence* σ is defined as a sequence of pairs $[(a_i, t_i)]$ where a_i is an action in the FSM, and t_i is the time event $e(a_i)$ occurs.

Definition 3: An action sequence $\sigma = [(a_i, t_i)]$ is *legal* if

- $\text{snk}(a_i) = \text{src}(a_{i+1})$;
- a_{i+1} is triggered by $e(a_{i+1})$ at time $t_{i+1} > t_i$, i.e. $t_{i+1} = (k + \lfloor \frac{t_i}{T_{e(a_{i+1})}} \rfloor) \times T_{e(a_{i+1})}$, $\forall k = \mathbb{N}^+$.

A legal action sequence $\sigma = [(a_i, t_i)]$ is a path in which a tight edge $(A_{i,t_i}, A_{i+1,t_{i+1}})$ connects any two consecutive pairs (a_i, t_i) and (a_{i+1}, t_{i+1}).

Definition 4: An action sequence $\sigma = [(a_i, t_i)]$ of length n is defined as *request-critical* for a time interval of length t if and only if

1) it is legal;
2) the last action is triggered within t time units from the activation of the first action, i.e. $t_n - t_1 < t$;
3) the total execution time $\sum_{i=1}^{n} C_{a_i}$ is the maximum among the sequences that satisfy 1) and 2).

This maximum execution time request is defined as the *request bound function* of the FSM $\mathcal{F}$ for the length t, denoted as $\mathcal{F}.rbf(t)$ (or simply $rbf(t)$) [8]. The definition of *demand-critical action sequence* and *demand bound function* over time t (denoted as $dbf(t)$) are similar, except that in condition 2) we require that the *deadline* of the last action is within t time units from the activation of the first action (frame separation property).

We now demonstrate that we can simplify the digraph by removing all the loose edges without changing the *rbf* and *dbf* functions.

Theorem 1: rbf and *dbf* do not change when all loose edges in the digraph task model are removed.

Proof: Suppose there is a request-critical action sequence $\sigma = [(a_i, t_i)]$ for an interval of length t with n actions in which some of them might be loose. We show that it is possible to construct another action sequence $\sigma' = [(a_i, t'_i)]$ entirely consisting of tight actions with the same *rbf* by the following iterative replacements.

- $t'_1 = t_1$;
- for $i = 1, ..., n-1$, define the next action instance as the same action in the original sequence, only triggered at a time $t'_{i+1} \leq t_{i+1}$ such that $(A_{i,t'_i}, A_{i+1,t'_{i+1}})$ is a tight edge in the digraph model.

It is easy to prove by induction that $t'_n \leq t_n$ which implies $t'_n - t'_1 \leq t_n - t_1 < t$. σ' is an action sequence with the same actions as σ (thus the same total execution times), thus it is also a request-critical action sequence for an interval of length t. In addition, all the edges in σ' are tight. A similar reasoning can be applied to the demand-critical action sequence. ■

The digraph obtained after removing all loose edges repeats every hyperperiod. Thus, it is sufficient to reason about the possible transitions and actions triggered by the events in one hyperperiod. For each event, we generate a set of vertices in the digraph, each vertex corresponds to the action triggered by the specific event. For example, for the two possible arrivals of event e_2 at time $0ms$ and $5ms$, we create two vertices $A_{2,0}$ and $A_{2,5}$ to represent the possible executions of a_2 within the hyperperiod. Figure 5 shows the reduced digraph model for the FSM $\mathcal{F}$ in Figure 1 (the back edges to $A_{2,0}$ and $A_{4,0}$ clearly refer to instances in the next hyperperiod). The minimum inter-arrival time labeled on edge $(A_{i,k}, A_{j,l})$ is $(l - k) \mod H_{\mathcal{F}}$ ($H_{\mathcal{F}} = 10$ in the example). In general, there are $\sum_{a_i} \frac{H_{\mathcal{F}}}{T_{e(a_i)}}$ vertices in the digraph model. The digraph in Figure 5 can give a more accurate *rbf* than the one in Figure 3, as shown in Figure 6. For example, *rbf* at $10ms$ is $1.3ms$, against a pessimistic estimate of $1.85ms$ using the model of Figure 3.

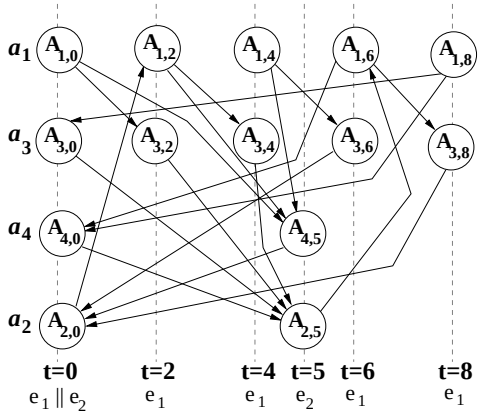

Figure 5. The simplified digraph task model of Figure 4

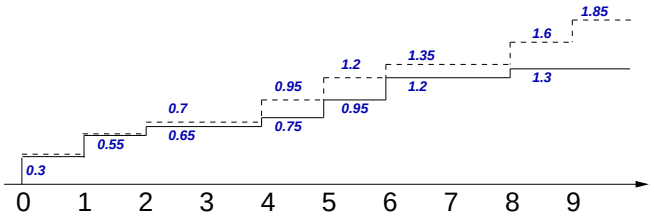

Figure 6. Request bound functions of the digraph models in Figure 3 (dotted line) and in Figure 5 (solid line)

Once the digraph model is generated, the analysis proposed in [19] can be used as a sufficient condition for checking the system schedulability. However, there are two sources of possible inaccuracy. The analysis in [19] assumes tasks have arbitrary offsets, while in synchronous FSMs events have the same offset. Also, [19] assumes earliest deadline first scheduling, while most commercial code generators for synchronous FSMs (e.g. [1] [2]) assume a conventional static-priority scheduling algorithm.

As for the efficiency of the schedulability analysis, there are two possible improvements compared to [19]. For synchronous FSMs, the possible event pattern repeats every hyperperiod, so are the demand and request bound functions. This leads to the introduction of the concept of execution request matrix, the request bound function for one hyperperiod (with the refinement by a pair of start and end states).

Finally, the algorithm of calculating *rbf* and *dbf* functions can be further improved by leveraging the periodicity of the execution request matrix (in max-plus algebra).

In the following section, we discuss the conditions for an exact analysis of task systems implementing synchronous FSMs, scheduled with fixed priority and preemption.

IV. Overview of Schedulability Analysis

In a Stateflow model, finite state machines are executed synchronously, that is, all trigger events and actions occur with the same offset. Without loss of generality, **all event offsets are assumed to be 0**. Therefore, the approach to schedulability analysis is the one typical of systems with static offsets, where feasibility is checked for a representative number of level-i busy periods. Theorem 2 in [21] defines the set of busy periods to be examined for periodic tasks with synchronized offsets. We first restate the concept of busy period as introduced in [15].

Definition 5: A *priority level-i busy period* is defined as follows [15]:

- It starts at some time s when a task of priority i or higher is ready for execution, and all the task instances with priority i or higher that are activated strictly before s have finished their execution (thus can be ignored).
- It is a continuous time interval during which any task of priority lower than i is unable to start execution.
- It ends at the earliest time f when there are no tasks of priority i or higher waiting to be executed that are queued strictly before time f.

The request bound function of an FSM over time t is the maximum cumulative execution times by transitions that are activated within *any interval* of length t. However, tasks implementing synchronous FSMs have static offsets. Therefore, we introduce the request and demand bound functions for a given time interval.

Definition 6: The *request bound function* of an FSM $\mathcal{F}$ during a time interval $\Delta = [s, f)$, (s *inclusive* and f *exclusive*), denoted as $\mathcal{F}.rbf(\Delta)$, is the maximum sum of execution times by the actions of $\mathcal{F}$ that have their activation time within Δ.

Definition 7: The *demand bound function* of an FSM $\mathcal{F}$ during the time interval $\Delta = [s, f]$ (both s and f are included in the interval), denoted as $\mathcal{F}.dbf(\Delta)$, is the maximum sum of execution times by the actions of $\mathcal{F}$ that have their activation time *and* deadline within Δ.

Similar to $rbf(t)$ and $dbf(t)$, both $rbf[s, s+t)$ and $dbf[s, s+t]$ are monotonically increasing with respect to t. By definition, these functions satisfy the following

$$rbf(t) = \max_s rbf[s, s+t), \quad dbf(t) = \max_s dbf[s, s+t]$$

Using these definitions, the following two theorems provide the condition for schedulability.

Theorem 2: Task i is schedulable in a level i busy period $[s, f)$ if $\forall t \in [s, f]$, $\exists t' \in [s, t]$ such that

$$\tau_i.dbf[s, t] + \sum_{j \in hp(i)} \tau_j.rbf[s, t') \leq t' - s$$

Theorem 3: The schedulability of a task i can be checked by examining the busy periods starting at the release of a task j of priority higher than or equal to task i.

Theorems 2 and 3 are trivial extensions of the corresponding ones in [8] and [21], respectively. They can be demonstrated in a very similar way, and we omit the proof here. Theorem 3 states that in order to guarantee schedulability, we must consider all level-i busy periods. Because of the periodicity of the trigger events, a busy period starts at an integer multiple of the event period within the system hyperperiod H. For example, consider a system including 2 blocks: a high priority Dataflow block with period 4, and the low priority Stateflow block in Figure 1 (with event periods $\{2, 5\}$). The start times of the busy periods to be considered are $\{2, 4, 5, 6, 8, 10, 12, 14, 15, 16, 18\}$.

The number of busy periods can be very large for Stateflow systems with many different event rates. If the possible periods of transitions include a maximum of $1000ms$ and a minimum of $5ms$, the typical runtime for the schedulability analysis is more than ten minutes. However, if we limit the set of periods to 5, 10, 20, 25, 50, and $100ms$, the runtime is reduced to be less than two seconds.

Checking the schedulability of task i in a busy period starting at s by Theorem 2 requires first to compute $rbf[s, t)$ and $dbf[s, t]$ for a given pair s, t. The solution to this problem will be answered in Section V. Also, we need to find the endpoint t on which to compute these functions. In Section VI, we derive a bound on the length of the busy period and therefore for the set of t values.

V. Calculation of $rbf(\Delta)$

In this section, we show how to compute $rbf(\Delta)$. The computation of $dbf(\Delta)$ can be easily derived from the procedure for $rbf(\Delta)$ by filtering the action execution times according to their deadlines. Because of the periodicity of the trigger events, if we move (left or right) the time interval Δ by an integer number of hyperperiods, the functions $rbf(\Delta)$ and $dbf(\Delta)$ remain the same. Formally, $\forall k \in \mathbb{N}$,

$$\begin{cases} \mathcal{F}.rbf[s + kH_{\mathcal{F}}, f + kH_{\mathcal{F}}) = \mathcal{F}.rbf[s, f) \\ \mathcal{F}.dbf[s + kH_{\mathcal{F}}, f + kH_{\mathcal{F}}] = \mathcal{F}.dbf[s, f] \end{cases} \quad (1)$$

Furthermore, the release times and absolute deadlines for the jobs of $\mathcal{F}$ can only belong to the *ordered* set $\mathbf{T}_{\mathcal{F}}$. Hence, we only need to consider the calculation of $rbf[s, f)$ and $dbf[s, f]$ for all $s, f \in \mathbf{T}_{\mathcal{F}}$, $s \in [0, H_{\mathcal{F}})$.

We now refine the concept of $rbf(\Delta)$ for a given pair of start and end states.

Definition 8: Given the start and end states s_i and s_j, the *request bound function* $\mathcal{F}.rbf_{i,j}(\Delta)$, is defined as the maximum accumulative execution times of any legal action sequence $[(a_k, t_k), k = 1, ..., n]$ of $\mathcal{F}$ such that

- the source state of the first transition is s_i;
- the sink state of the last transition is s_j;
- $t_1 \geq s$ and $t_n < f$;
- $\mathcal{F}.rbf_{i,j}(\Delta) = \max(\sum_{k=1}^{n} C_{a_k})$.

If s_j is not reachable from s_i, then $\mathcal{F}.rbf_{i,j}(\Delta)$ is defined as $-\infty$. With this definition, the domain for the possible *rbf* and *dbf* values is $\mathbb{R}^* = \mathbb{R} \bigcup \{-\infty\}$. By these definitions,

$$\mathcal{F}.rbf(\Delta) = \max_{i,j} \mathcal{F}.rbf_{i,j}(\Delta). \quad (2)$$

Also, $rbf_{i,j}(\Delta)$ is additive, i.e. $\forall i, j, \forall t \in [s, f]$,

$$\mathcal{F}.rbf_{i,j}[s, f) = \max_{m}(\mathcal{F}.rbf_{i,m}[s, t) + \mathcal{F}.rbf_{m,j}[t, f)) \quad (3)$$

Thus, $rbf_{i,j}[s, f)$ for a long interval $[s, f)$ can be computed from its values for shorter intervals $[s, t)$ and $[t, f)$. Techniques from dynamic programming can be used for an efficient calculation of $rbf_{i,j}(\Delta)$. In addition, in the remainder of this section we show how results from max-plus algebra (which has tight connection to dynamic programming) can be leveraged to find its periodicity. The $rbf(\Delta)$ function (without constraints on the initial and final states) is not additive, as shown by the following equation, thus dynamic programming techniques and max-plus algebra results cannot be directly used to speedup its computation.

$$\begin{aligned} & \mathcal{F}.rbf[s, t) + \mathcal{F}.rbf[t, f) \\ = & \max_{i,k} \mathcal{F}.rbf_{i,k}[s, t) + \max_{l,j} \mathcal{F}.rbf_{l,j}[t, f) \\ = & \max_{i,j,k,l} (\mathcal{F}.rbf_{i,k}[s, t) + \mathcal{F}.rbf_{l,j}[t, f)) \\ \geq & \max_{i,j,k=l} (\mathcal{F}.rbf_{i,k}[s, t) + \mathcal{F}.rbf_{l,j}[t, f)) = \mathcal{F}.rbf[s, f) \end{aligned}$$

The computation of $rbf_{i,j}(\Delta)$ requires searching the reachable states within the possible sequences of events. For each state s_i, we generate an infinite sequence of vertices in the *reachability graph*. Each vertex S_{i,t_i^-} corresponds to the source state s_i before any action triggered by the event at time t_i, $\forall t_i \in \mathbf{T}_{\mathcal{F}}$. The edge $(S_{i,t_i^-}, S_{j,t_{i+1}^-})$ is added when:

- t_{i+1} is the next time instant after t_i in the set $\mathbf{T}_{\mathcal{F}}$;
- there is a transition $\theta_k = \{s_i, s_j, e_{\theta_k}, g_k, a_k, p_k\} \in \mathbf{\Theta}$ from s_i to s_j in the FSM; the edge is labeled with C_{a_k}.
- t_i is an integer multiple of $T_{e(a_k)}$.

Intuitively, edge $(S_{i,t_i^-}, S_{j,t_{i+1}^-})$ corresponds to a transition from s_i to s_j at time t_i. Furthermore, stuttering edges from S_{i,t_i^-} to S_{i,t_{i+1}^-} (denoted by the symbol $\perp$ and labeled with 0) are added to represent the possible stuttering behavior. The reachability graph of the example is shown in Figure 7.

Every path from S_{i,t_i^-} to S_{j,t_j^-} in the reachability graph corresponds to a possible legal action sequence from s_i to s_j during the interval $[t_i, t_j)$. Thus, $rbf_{i,j}[t_i, t_j)$ can be computed as the longest path from S_{i,t_i^-} to S_{j,t_j^-}. This problem can be solved in cubic time to the size of the graph (which is linear to the length of the time interval) by negating the weight of the edges and leveraging the classic Floyd-Warshall algorithm [12] for all-pairs shortest path problem.

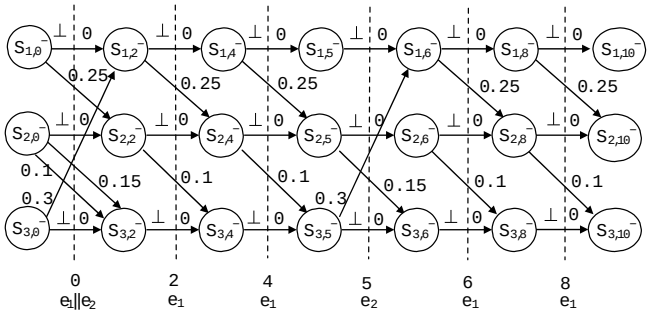

Figure 7. Reachability graph of the FSM in the first hyperperiod

Figure 8 shows the $rbf[0, 10)$ function in the first hyperperiod, where the worst case *rbf* is actually obtained when starting from s_3. As a comparison, $rbf(10)$ for any time interval of length $10ms$ is also shown in the figure.

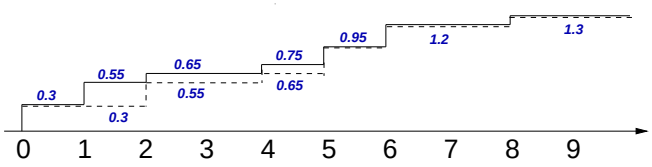

Figure 8. $rbf[0, 10)$ (dotted line) and $rbf(10)$ (solid line)

Clearly, it is inefficient to build the reachability graph for arbitrarily long time intervals. In the following, we show how to utilize the repeating pattern of trigger events in the hyperperiod as well as the periodicity of *rbf* function to efficiently compute it for any interval. *The asymptotic complexity of the resulting algorithm does not depend on the length of the time interval, but is only a function of the number of states in the FSM.*

A. The execution request matrix

For simplicity, we denote the request bound function within k hyperperiods for a pair of given start and end states as $x_{i,j}^{(k)} = rbf_{i,j}[0, kH)$, and the $n \times n$ matrix $\mathbf{X}^{(k)}(\mathcal{F}) = (x_{i,j}^{(k)}, i, j = 1, ..., n)$ where $n = |\mathbf{S}|$ is the number of states in the FSM. We define $\mathbf{X}^{(1)}(\mathcal{F})$ as *the execution request matrix* of the FSM $\mathcal{F}$, and simply denote it as $\mathbf{X}$. As a special case of Equation (3), we have

$$\forall i, j, \forall 1 \leq l < k \quad x_{i,j}^{(k)} = \max_{m}(x_{i,m}^{(l)} + x_{m,j}^{(k-l)}) \quad (4)$$

Equation (4) can be used to compute the *rbf* function for intervals spanning multiple hyperperiods instead of building large reachability graphs. This results in a significant improvement, typically one to two orders of magnitudes. *The same speedup cannot be applied using generic digraph task models and is not considered in the analysis in [19].*

Example: For the example FSM in Figure 1, we consider the action sequence of $[(a_1, 0), (a_3, 2), (a_2, 5), (a_1, 6),$

$(a_3, 8)]$. The total execution time request is $1.0ms$, and both the start state and end state are s_1. It can be verified that no other action sequence with s_1 as a start and end state has a cumulative execution time larger than $1.0ms$. Thus $x_{1,1}^{(1)} = 1.0ms$. The matrix $\mathbf{X}^{(1)}$ is calculated as

$$\mathbf{X}^{(1)} = \begin{bmatrix} 0.65 & 0.9 & 1.0 \\ 0.45 & 0.7 & 0.8 \\ 0.95 & 1.2 & 1.3 \end{bmatrix}$$

Moreover, it can be verified that $\forall k \in \mathbb{N}$, matrix $\mathbf{X}^{(k+1)}$ is

$$\mathbf{X}^{(k+1)} = \begin{bmatrix} 0.65 & 0.9 & 1.0 \\ 0.45 & 0.7 & 0.8 \\ 0.95 & 1.2 & 1.3 \end{bmatrix} + k \times \begin{bmatrix} 1.3 & 1.3 & 1.3 \\ 1.3 & 1.3 & 1.3 \\ 1.3 & 1.3 & 1.3 \end{bmatrix}$$

This example discloses some periodic behavior in the worst case execution request for the FSM, even if the task graph does not have a cyclic structure as defined in the literature, e.g. [8] [9] [18]. This periodicity property is applicable to any FSM system. In the next subsection, we introduce the max-plus algebra and related research results that can be leveraged to compute the request bound function for large time intervals using $\mathbf{X}^{(k)}$.

B. Max-plus algebra and periodic matrix power sequence

The matrix $\mathbf{X}^{(k)}$ and its elements $x_{i,j}^{(k)}$ are defined over the domain $\mathbb{R}^* = \mathbb{R} \bigcup \{-\infty\}$. A max-plus algebra [5] is defined over $\mathbb{R}^*$ (or in general, any dioid) with operations maximum (denoted by the max operator $\oplus$) and addition (denoted by the plus operator $\otimes$) defined as

$$x \oplus y = \max(x, y), \quad x \otimes y = x + y \tag{5}$$

It is easy to verify that $-\infty$ is neutral with respect to $\oplus$, i.e. $x \oplus (-\infty) = x, \forall x \in \mathbb{R}^*$. Likewise, 0 is neutral with respect to $\otimes$, $x \otimes 0 = x, \forall x \in \mathbb{R}^*$.

The matrix operations over $\mathbb{R}^*$ are defined in the same way as the matrix operation over any field. For example, $\mathbf{Z} = \mathbf{X} \oplus \mathbf{Y}$ is defined by taking the maximum operation of the corresponding elements of the matrices $\mathbf{X}$ and $\mathbf{Y}$, i.e. $z_{i,j} = x_{i,j} \oplus y_{i,j}$.

For matrices $\mathbf{X} \in \mathbb{R}^*(m, k)$ and $\mathbf{Y} \in \mathbb{R}^*(k, n)$, the result of the multiplication is a matrix $\mathbf{Z} \in \mathbb{R}^*(m, n)$, where its element is

$$z_{i,j} = \max_{l=1}^{k}(x_{i,l} + y_{l,j}) \tag{6}$$

Under max-plus algebra, Equation (4) can be rewritten as $\mathbf{X}^{(k)} = \mathbf{X}^{(l)} \otimes \mathbf{X}^{(k-l)}$ (max-plus multiplication). Thus $\mathbf{X}^{(k)}$ is the k-th power of the matrix $\mathbf{X}$, with elements $x_{i,j}^{(k)}$.

The following definitions and results provide insight on how to partition the computation of long intervals into a recurrent part plus some possible initial terms (as for the powers of $\mathbf{X}$ in the example).

Definition 9: A sequence $x^* = (x^{(r)}), r \in \mathbb{N}^+$ is defined as *almost linear periodic* if there exists a finite real number $q \in \mathbb{R}$ and a pair of integers d and p such that

$$\forall r > d, \;\; x^{(r+p)} = x^{(r)} + p \times q \tag{7}$$

The smallest number p with the above properties is called the *linear period* of x^*, denoted as $p = lper(x^*)$. q is called the *linear factor* of x^*, denoted as $q = lfac(x^*)$. Finally, the smallest number d with the above properties is called the *linear defect* (or *coupling time*), denoted as $d = ldef(x^*)$.

Definition 10: The matrix $\mathbf{X} = (x_{i,j})$ is defined as *almost linear periodic* if for each element $x_{i,j}$ in its power sequence $\mathbf{X}^* = (\mathbf{X}^{(r)}), r \in \mathbb{N}^+$, the sequence $x_{i,j}^*$ is almost linear periodic. The matrix $lfac(\mathbf{X}^*) = (lfac(x_{i,j}^*))$ is the *linear factor* of $\mathbf{X}^*$, the number $ldef(\mathbf{X}) = \max\{ldef(x_{i,j}^*)\}$ is the *linear defect* of $\mathbf{X}$, and $lper(\mathbf{X}) = lcm\{lper(x_{i,j}^*)\}$ is the *linear period* of $\mathbf{X}$.

The properties of a matrix $\mathbf{X}$ are studied by its corresponding digraph.

Definition 11: The *digraph* $\mathcal{G}(\mathbf{X})$ of a square matrix $\mathbf{X} \in \mathbb{R}^*(n, n)$ is a weighted digraph (V, E, w) with the set of nodes $V = \{1, ..., n\}$. The set of arcs $E = \{(i, j)\}$ for every finite $x_{i,j}$ of matrix $\mathbf{X}$ is weighted by the corresponding $x_{i,j}$. If $x_{i,j} = -\infty$, there is no arc from i to j in $\mathcal{G}(\mathbf{X})$.

A *path* π in $\mathcal{G}$ is a sequence of nodes $(i_1, i_2, ..., i_{r+1})$ such that (i_k, i_{k+1}) is an arc in E. The *length* $|\pi|$ of π is r. If $i_1 = i_{r+1}$, π is called a *cycle*. The *weight* of π, denoted as $w(\pi)$, is defined as the sum of the weights of its arcs.

For a cycle c, the *cycle mean*, denoted as $\bar{w}(c)$, is defined as the ratio between its weight and its length, i.e. $\bar{w}(c) = w(c)/|c|$. The maximum mean of any cycle in $\mathcal{G}(\mathbf{X})$ is $\lambda(\mathbf{X}) = \max_{c \in \mathcal{G}} \bar{w}(c)$. $\mathcal{G}(\mathbf{X})$ is called strongly connected if all nodes are connected inside a common cycle. In this case, $\mathbf{X}$ is called *irreducible*.

Definition 12: A subgraph $\mathcal{K} = (K, E \bigcap K \times K)$ of $\mathcal{G}(\mathbf{X})$ is a *highly connected component* if all its nodes are contained in a cycle with mean equal to $\lambda(\mathbf{X})$. Its *high period* is defined as $hper(\mathcal{K}) = gcd\{|c| : c \text{ is a cycle in } \mathcal{K}, \bar{w}(c) = \lambda(\mathbf{X})\}$. The set of all highly connected components of $\mathcal{G}(\mathbf{X})$ is denoted as $HCC^*(\mathcal{G}(\mathbf{X}))$.

Irreducible matrices can be analyzed using the following results. In [5] it is demonstrated that an irreducible matrix is also almost linear periodic. The linear defect of an irreducible matrix can be upper bounded by, for example, algorithms in [14] in $O(n^3)$ time. In [13] an $O(n^3)$ algorithm is proposed for computing the linear period and linear factor of an irreducible matrix, based on the following theorem.

Theorem 4: If $\mathbf{X} \in \mathbb{R}^*(n, n)$ is irreducible, then [13]

- $\mathbf{X}$ is almost linear periodic;
- $lfac(\mathbf{X}) = Q$, with $q_{i,j} = \lambda(\mathbf{X})$ for all i,j;
- $lper(\mathbf{X}) = lcm\{hper(\mathcal{K}) : \mathcal{K} \in HCC^*(\mathcal{G}(\mathbf{X}))\}$.

Unfortunately, if the matrix is reducible, the matrix could still be almost linear periodic, but deciding whether this is the case has been demonstrated to be NP-complete [13]. When the matrix is not almost linear periodic, there still can be a way to avoid computing the *rbf* function over long time intervals by detecting a more general type of periodicity. Molnárová [17] introduces the concept of general periodicity where linear periodicity is a special case.

Definition 13: A sequence $x^* = (x^{(r)})$, $r \in \mathbb{N}^+$ is defined as *almost generally periodic* if there exists a pair of integers d and p and a set of (not necessarily finite) numbers $q(k) \in \mathbb{R}^*, k = 1, ..., p$ such that

$$\forall k = 1, ..., p, \forall r > d, r \equiv k \bmod p, \quad x^{(r+p)} = x^{(r)} + p \times q(k)$$

The smallest number p (d) with the above properties is called the *generalized period* (*generalized defect*) of x^*, denoted as $p = gper(a^*)$ ($d = gdef(x^*)$). q is called the *generalized factor* of x^*, denoted as $q = gfac(x^*)$.

Definition 14: The matrix $\mathbf{X} = (x_{i,j})$ is defined as *almost generally periodic* if for each element $x_{i,j}$ in its power sequence $\mathbf{X}^* = (X^{(r)}), r \in \mathbb{N}^+$ the sequence $x^*_{i,j}$ is almost generally periodic. The matrix $gfac(\mathbf{X}^*) = (gfac(x^*_{i,j}))$ is called the generalized factor matrix of $\mathbf{X}$, the number $gdef(\mathbf{X}) = \max gdef(x^*_{i,j})$ is called its generalized defect, and $gper(\mathbf{X}) = lcm\{gper(x^*_{i,j})\}$ is its generalized period.

The following theorem states the periodicity property is applicable for every matrix.

Theorem 5: Every matrix over a max-plus algebra is almost generally periodic [17].

However, we still need to compute the $gper$ and $gfac$ terms, and the problem of computing these terms is shown to be NP-hard [17]. Nevertheless, the complexity is a function of the size of the matrix (or the corresponding digraph), i.e., the number of states in the FSM, but is asymptotically independent from the power of the matrix (or the length of Δ for which $rbf(\Delta)$ function is computed). Therefore, it is still possible to compute these functions for small FSMs and large Δ. We leave the analysis of the tradeoffs among different approaches to future work.

From now on, we assume that given the finite state machine $\mathcal{F}$, the $n \times n$ matrix $\mathbf{X} = (x_{i,j} = rbf^{[0,H)}_{i,j}, i, j = 1, ..., n)$ where $n = |\mathbf{S}|$ can be calculated. Its generalized defect $d = gdef(\mathbf{X})$, generalized period $p = gper(\mathbf{X})$, and generalized factor matrix $q = gfac(\mathbf{X})$ are also assumed to be computed for the following discussion.

C. Calculation of $rbf[s, f)$ for small f

We denote $n_s = \lceil \frac{s}{H} \rceil$ and $n_f = \lfloor \frac{f}{H} \rfloor$. We discuss how to calculate the values of the functions for relatively small f when $n_f - n_s \leq gdef(\mathbf{X})$. As a special case of (3), the *rbf* function can be decomposed into functions defined over the intervals $[s, n_sH)$, $[n_sH, n_fH)$, and $[n_fH, f)$.

$$\begin{aligned} rbf_{i,j}[s, f) = & \max_{k,l}(rbf_{i,k}[s, n_sH) + rbf_{k,l}[n_sH, n_fH) \\ & + rbf_{l,j}[n_fH, f)) \\ = & \max_{k,l}(rbf_{i,k}[s, n_sH) + x^{(n_f - n_s)}_{k,l} \\ & + rbf_{l,j}[0, f - n_fH)) \end{aligned} \tag{8}$$

The value of $\mathbf{X}^{(n_f - n_s)}$ can be computed using (4). $rbf[s, f)$ is computed as the maximum among all the $rbf_{i,j}[s, f)$ for all possible pairs i and j, as defined in Equation (2).

D. Calculation of $rbf[s, f)$ for large f

For $n_f - n_s > gdef(\mathbf{X})$, Equation (8) still applies, but $x^{(n_f - n_s)}_{k,l}$ can be directly computed using its periodicity property ($d = gdef(\mathbf{X})$, $p = gper(\mathbf{X})$, $Q = gfac(\mathbf{X})$).

$$x^{(n_f - n_s)}_{k,l} = x^{(n)}_{k,l} + (n_f - n_s - n) \times q_{k,l}(k)$$
where $n \leq d$ and $n + p > d$, $\quad n_f - n_s \equiv n \equiv k \mod p$.

VI. Upper bounds on *rbf* and *dbf*

In this section, we derive a bound for the *rbf* and *dbf* functions. This bound can be used to compute an upper bound for the busy period in the case of static priority scheduling (see Section IV). We first define the utilization of an FSM, which describes the asymptotical maximum execution rate that the FSM can create and then we derive the bounds for *rbf* and *dbf*.

Definition 15: The utilization of an FSM, denoted as $U(\mathcal{F})$ is defined as the maximum cycle mean of its execution request matrix $\mathbf{X}(\mathcal{F})$ divided by its hyperperiod, i.e.

$$U(\mathcal{F}) = \frac{\lambda(\mathbf{X}(\mathcal{F}))}{H_{\mathcal{F}}}.$$

Theorem 6: The *rbf* and *dbf* functions of FSM $\mathcal{F}$ are bounded by (where $C^{\text{sum}} = \sum_{\theta_j \in \mathbf{\Theta}} C_{a_j} \times \frac{H_{\mathcal{F}}}{T_{e(a_j)}}$)

$$\begin{aligned} \mathcal{F}.dbf[s, f] &\leq (f - s)U(\mathcal{F}) + C^{\text{sum}} \\ \mathcal{F}.rbf[s, f) &\leq (f - s)U(\mathcal{F}) + 2C^{\text{sum}} \end{aligned} \tag{9}$$

Proof: Since the relative deadlines of the actions are no greater than $H_{\mathcal{F}}$, and C^{sum} is the upper bound on the total execution request within one hyperperiod $H_{\mathcal{F}}$, it is

$$\forall t, \mathcal{F}.rbf[s, f) \leq \mathcal{F}.dbf[s, f + H_{\mathcal{F}}] \leq \mathcal{F}.dbf[s, f] + C^{\text{sum}}.$$

The bound on the *dbf* function can be proved using the results from [19] on digraph task model. An FSM can be transformed to an equivalent digraph task model as described in Section III, where the corresponding digraph model contains $\frac{H_{\mathcal{F}}}{T_{e(a_j)}}$ nodes for each action a_j, and the sum of the execution times of all nodes in this digraph task model is $\sum_{\theta_j \in \mathbf{\Theta}} C_{a_j} \times \frac{H_{\mathcal{F}}}{T_{e(a_j)}} = C^{\text{sum}}$. By Lemma V.2 from [19], $\mathcal{F}.dbf[s, f] \leq (f - s)U(\mathcal{F}) + C^{\text{sum}}$. ■

From these upper bounds on *rbf* and *dbf*, an upper bound on the length of the busy periods can be computed. For an unschedulable task i, there exists a busy period $[s, f)$ and $t \in [s, f)$ such that $\forall t' \in [s, t]$, $\tau_i.dbf[s, t] + \sum_{j \in hp(i)} \tau_j.rbf[s, t') > t' - s$ holds. In particular, this must be true for the endpoint $t' = t$. By (9) and simple math, an upper bound on the time instant t to be checked for schedulability can be derived as

$$t < s + \frac{C^{\text{sum}}_i + 2\sum_{j \in hp(i)} C^{\text{sum}}_j}{1 - U_i - \sum_{j \in hp(i)} U_j}.$$

VII. Experimental evaluation

We generate 1000 random systems with 20 Stateflow blocks, each block with a maximum of 15 states and random transitions between them. The maximum degree of the blocks is 21, and the average is 3.2. A period of 5, 10, 20, 25, 50, or 100ms is randomly assigned to each transition. The density (defined as the utilization scaled by the ration between the hyperperiod of the Stateflow and its deadline) is uniformly distributed between 0 and 100%. The system utilization is very low (around 7.2%), as the shortest relative deadline of the actions is the greatest common divisor of the event periods, which is typically much shorter than the hyperperiod of the Stateflow block. 419 of these 1000 systems are schedulable using the analysis assuming synchronized offsets and deadline monotonic priority assignments.

The first objective of our experiments is to evaluate the possible improvements with respect to analysis accuracy compared to methods based on digraph task models. To compare the two methods on equal terms, we use deadline monotonic to assign fixed priorities to the tasks in the digraph model and we use the static priority analysis method instead of the one for earliest deadline first policy. Our analysis method finds 61 schedulable task sets that are evaluated as infeasible by the digraph task formalization and analysis method (or a 6.1% false negative schedulability from the analysis for digraph tasks). This pessimism comes from the assumption of arbitrary task offsets in the analysis of digraph task model.

Next, we tried to evaluate the possible benefits in terms of running time resulting from the use of the periodicity property for the *rbf* and *dbf* functions. Among 20000 total randomly generated Stateflow blocks, only 1861 are irreducible. We leverage the property of linear periodicity to efficiently calculate the power sequence of the execution request matrix for these blocks. This improvement results in a reduction on the runtime of schedulability analysis, from a total of 137.7 seconds to 134.2 seconds. These savings come from the scenarios that the busy period for schedulability analysis is several times longer than the linear period of the execution request matrix for irreducible Stateflow blocks, where this linear periodicity can help speed up the calculation of *rbf* and *dbf* functions by about three times.

VIII. Conclusions and Future Work

In this paper we present methods for the schedulability analysis of systems with synchronous finite state machines. We discuss similarities and differences with respect to current task graph models, and present analysis techniques that can benefit from the periodicity of the *rbf* and *dbf* functions. The asymptotic complexity of the resulting algorithm is independent from the length of the time interval, but only a function of the number of states in the FSM. As for future work, we plan to explore efficient algorithms for the case that the execution request matrix of the Stateflow block is reducible. We also consider the extension of the periodicity of *rbf* and *dbf* functions to generic digraph task models.

Finally, we would like to thank Prof. Sanjoy Baruah for the discussion and suggestions on task models.

References

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

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

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

[4] M. Anand, "Conditional models for compositional design of real-time embedded systems," Ph.D. dissertation, University of Pennsylvania, Janurary 2008.

[5] F. Baccelli, G. Cohen, G. Olsder, and J. Quadrat, *Synchronization and Linearity: An Algebra for Discrete Event Systems*. Wiley, 1992.

[6] S. Baruah, D. Chen, S. Gorinsky, and A. Mok, "Generalized multiframe tasks," *Real-Time Syst.* 17(1): 5–22, 1999.

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

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

[9] S. Baruah, "The non-cyclic recurring real-time task model," in *Proc. the 31st IEEE Real-Time Systems Symposium*, 2010.

[10] 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.

[11] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *International Journal of Information and Computation* 205(8): 1149–1172, August 2007.

[12] R. Floyd, "Algorithm 97: Shortest Path," *Communications of the ACM* 5(6): 345, June 1962.

[13] M. Gavalec, "Linear matrix period in max-plus algebra," *Linear Algebra and its Applications* 307(1-3): 167–182, 2000.

[14] M. Hartmann and C. Arguelles, "Transience bounds for long walks," *Math. Oper. Res.* 24: 414–439, May 1999.

[15] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," in *Proc. IEEE Real-Time Systems Symposium*, 1990.

[16] A. K. Mok and D. Chen, "A multiframe model for real-time tasks," in *Proc. IEEE Real-Time Systems Symposium*, 1996.

[17] M. Molnárová, "Generalized matrix period in max-plus algebra," *Linear Algebra and its Applications*, 404: 345–366, 2005.

[18] N. T. Moyo, E. Nicollet, F. Lafaye, and C. Moy, "On schedulability analysis of non-cyclic generalized multiframe tasks," in *Proc. the 22nd Euromicro Conference on Real-Time Systems*, 2010.

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

[20] M. Stigge, P. Ekberg, N. Guan, , and W. Yi, "On the Tractability of Digraph-Based Task Models," in *Proc. the 23rd Euromicro Conference on Real-Time Systems*, 2011.

[21] K. Tindell, "Adding time-offsets to schedulability analysis," *Department of Computer Science, University of York, Report No. YCS-94-221*, 1994.

[22] Edward A. Lee and Pravin Varaiya, "Structure and Interpretation of Signals and Systems," Second Edition, LeeVaraiya.org, ISBN 978-0-578-07719-2, 2011.

[23] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", *Sci. Comput. Programming* 8(3):231-274, 1987.