Robust and Extensible Task Implementations of Synchronous Finite State Machines

Qi Zhu
UC Riverside
qzhu@ee.ucr.edu

Peng Deng
UC Riverside
pdeng002@ucr.edu

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

Haibo Zeng
McGill University
haibo.zeng@mcgill.ca

***Abstract*—Model-based design using synchronous reactive (SR) models is widespread for the development of embedded control software. SR models ease verification and validation, and enable the automatic generation of implementations. In SR models, synchronous finite state machines (FSMs) are commonly used to capture changes of the system state under trigger events. The implementation of a synchronous FSM may be improved by using multiple software tasks instead of the traditional single-task solution. In this work, we propose methods to quantitatively analyze task implementations with respect to a breakdown factor that measures the timing robustness, and an action extensibility metric that measures the capability to accommodate upgrades. We propose an algorithm to generate a correct and efficient task implementation of synchronous FSMs for these two metrics, while guaranteeing the schedulability constraints.**

I. Introduction

Model-based design based on synchronous reactive (graphical or textual) languages is increasingly used in the development of control algorithms, for the opportunity of early validation and verification coming from simulation and formal verification of properties. Today, the most popular tools supporting SR modeling are SCADE and Simulink. In both languages, the system is modeled as a network of dataflow and finite state machine blocks (called Stateflow in Simulink). At the end of the modeling stage, both environments offer tools for the automatic generation of a software or firmware implementation. The languages supported by SCADE (Esterel, Lustre, and SyncCharts [3]) are typically implemented as a single executable that runs according to an event server model. Reactions are decomposed into atomic actions that are partially ordered by the causality analysis of the program. The scheduling is generated at compile time, and the generated code executes without the need of an operating system. References to the code generation process can be found in [10, 15].

Similarly, in the implementation of Simulink models, current tools such as the Embedded Coder/Simulink Coder [1] generate a single periodic task for each Stateflow block (FSM). Every time this task is activated, it checks for active trigger events and processes them. To make sure it will not miss any trigger event, the task is executed at the greatest common divisor (GCD) of the periods of its trigger events. In [8], it is observed that such a single-task implementation may lead to unnecessary activations, and therefore reduce the system schedulability and cause memory overhead on communication buffers. By generating multiple software tasks according to a *partitioning* of the FSM based on the periods of the trigger events, the system schedulability and memory usage may be improved. Different task implementation models are discussed. However, no approach is provided for quantitatively comparing them. The design of a memory-efficient implementation of communication among blocks can be found in [7] and [6]. In [16], the schedulability analysis of Stateflow models or real-time task models derived by synthesizing a state machine formalism is discussed.

In this work, we propose methods to quantitatively analyze and compare task implementations with respect to a *breakdown factor* and an *action extensibility* metric. The two metrics have different practical implications, and might lead to different optimal solutions.

The breakdown factor is defined based on the concept of *breakdown utilization* (introduced in [12]) as the maximum scaling factor of task execution times that allows to retain feasibility (in our work, the factor applies to *actions* in FSMs). Intuitively, a larger breakdown factor allows a wider selection of implementation platforms (e.g. different processor speeds for cost/performance trade-off) and makes the system more robust with respect to timing variations.

Action extensibility follows the definition of *task extensibility* (proposed in [17]) as the maximum amount by which the execution time of each *single* action in the FSMs may increase without violating the system timing constraints. Optimizing action extensibility allows adding future functionality or upgrading existing ones without a major redesign cycle. This is imperative for large-volume and long-lifetime systems.

The number of possible multi-task implementations for a given FSM is exponential in the number of transitions. For complex automotive and avionics systems, the FSM may contain hundreds or thousand of states, which leads to a huge number of potential implementations. While intuitively generating more tasks provides more flexibility in scheduling, it also causes context switching overhead and in some cases leads to communication overhead. Furthermore, careful analysis is required to guarantee that the multi-task implementation is functionally correct.

The main contributions of this work include:

- algorithms for computing the breakdown factor and the action extensibility for systems implementing synchronous FSMs.

978-3-9815370-0-0/DATE13/ © 2013 EDAA

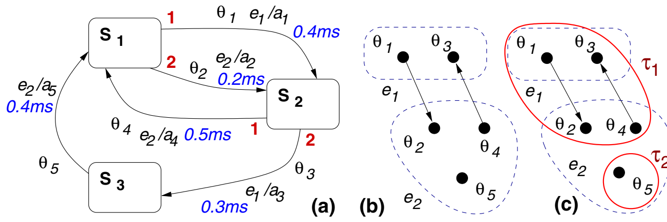

Fig. 1. An Example of FSM Representation

- an algorithm to improve the task implementations of synchronous FSMs based on these two metrics, while guaranteeing functional correctness and satisfying schedulability constraints.

The paper is organized as follows. Section II introduces models for synchronous FSMs and their task implementations, and presents a motivating example. Section III presents methods to compute the breakdown factor and the action extensibility for given task implementations. Section IV presents the algorithm to explore the task implementations according to the two metrics. Section V shows the experimental results and Section VI concludes the paper.

II. System Models and Motivation

A. Synchronous FSM model

We use the similar notations as in [8] to represent a synchronous (Mealy type, inspired by Stateflow) finite state machine. An FSM is defined by a tuple $(\mathbf{S}, S_0, \mathbf{I}, \mathbf{O}, \mathbf{E}, \mathbf{T})$, where $\mathbf{S} = \{S_0, S_1, S_2, \ldots S_l\}$ is a set of *states*, $S_0 \in \mathbf{S}$ is the *initial state*, $\mathbf{I} = \{i_1, i_2, \ldots i_p\}$ and $\mathbf{O} = \{o_1, o_2, \ldots o_q\}$ are the input and output *signals*. $\mathbf{E}$ is a set of *trigger (or activation) events*. Each event e_j is generated by the value change of a signal, and may only occur with a period t_{e_j} that is an integer multiple of a system *base period* t_{base} (i.e. $t_{e_j} = k_{e_j} \cdot t_{base}$). At each time instant $k \cdot t_{e_j}$, the event may or may not present (in which case the system may stutter). Finally, $\mathbf{T}$ is a set of transition rules. Each transition $\theta_j \in \mathbf{T}$ is a tuple $\theta_j = \{S_{s_j}, S_{d_j}, e_{\theta_j}, g_j, a_j, \gamma_j\}$, where S_{s_j} is the source state, S_{d_j} is the destination state, $e_{\theta_j} \in \mathbf{E}$ is the trigger event (different transitions may be triggered by the same event), g_j is the guard condition, a_j is the *action*, and γ_j is the evaluation order among the transitions that start from the same state. When two or more transitions from the same source state are enabled at the same time, the transition with the lowest order is taken.

Fig. 1(a) shows an example FSM. The periods of e_1 and e_2 are $2ms$ and $3ms$ respectively. For each transition, the associated trigger event and action are shown, along with the transition priority (in red and bold) and the execution time of the action (in blue and italic). The guard condition is omitted.

In this paper, we do not consider the case in which transitions are activated by a logical expression evaluated on multiple events, or events generated by transitions. The Stateflow semantics also allow more advanced constructs such as concurrent states, superstates, entry/exit actions, while actions and join transitions. In several cases, a hierarchical FSM can be flattened and our methods can apply. The conditions for the translation into flat FSMs are outlined in [14] and the composition rules can be found in [11].

B. Task implementation model

In this work, we consider implementing a set of FSMs $\mathcal{F} = \{F_1, F_2, \ldots F_m\}$ on a single embedded processor with a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots \tau_n\}$. Each task τ_k has period ψ_k and priority π_k. If a transition θ_j (and the corresponding action a_j) is implemented in τ_k, it is denoted as $\mu(\theta_j) = \tau_k$. We assume that only transitions from the same FSM may be implemented in the same task.

In synchronous FSMs, every transition occurs in *zero logical time*, meaning that it completes before the next event is processed. This assumption is important for determining the functional behavior of FSMs, which may be captured by a stream of input and output signal values. When implementing the synchronous FSMs with tasks, it is necessary to preserve the same stream of input and output signal values (*flow preservation*), which requires maintaining the same *logic order* among events and actions during execution. This means that in the implementation, executing in real *physical time*, any action execution and update of system state need to be completed before the next set of inputs is *processed* (by the task activated by the next event).

1) Single-task implementation: In a single-task implementation, all transitions from the same FSM F_k are implemented in a single task τ_k, i.e. $\forall \theta_j \in F_k, \mu(\theta_j) = \tau_k$. This is the approach adopted by the the Simulink Coder generator, and is called *baseline implementation* in [8]. The period of τ_k is the GCD of the periods of all trigger events, i.e. $\psi_k = GCD(t_{e_{\theta_j}})$ where $\theta_j \in F_k$. To ensure that every transition completes before the next set of inputs is processed (by a new instance of the same task), the largest action execution time should be no larger than the period, i.e. $\max(C_{a_j}) \leq \psi_k$ where C_{a_j} is the execution time of a_j.

2) Multi-task implementation: In [8], three task models are presented for multi-task implementations of synchronous FSMs – the partitioned model, the mixed-partitioned model and the deferred output update model. The deferred output update model is based on separating the *state update* function and the *output update* function into different tasks. This may require significant development and verification efforts and cause extra overhead if the two functions share significant amount of computation. Therefore, in this work, we focus on the partitioned model and the mixed-partitioned model, and generalize them to a *general partitioned model*.

To explain the task models, we define a *transition graph* G_k for each FSM F_k, where each transition is represented by a node and there is a directed edge between two nodes/transitions θ_i and θ_j when the source state of θ_i and θ_j is the same and the order of θ_i is lower than the order of θ_j. A *partition* P divides all the nodes in G_k into a *partition graph* $G_k(P)$, where each partition set p_i is represented by a node v_i, and a directed edge exists between two nodes v_i and v_j if at least one transition in p_i has lower order than

another transition in p_j. Fig. 1(b) shows a special partition graph $G_k(P_e)$ that is based on the *event partition* P_e, where the nodes/transitions are divided based on their events.

A multi-task implementation of the FSM F_k consists of a partition P, where the transitions in each set p_i is implemented by a task τ_i. We assume a fixed priority scheduling among tasks, therefore a feasible task implementation needs to be consistent with the transition priorities, i.e. τ_i has higher priority than τ_j if a transition in τ_i has lower order than a transition in τ_j. During execution, τ_i sends an inhibition signal to all lower priority tasks τ_j generated from the same FSM, if one of its transitions executes. When this happens, τ_j skips. It is easy to see that a task implementation has a feasible priority assignment only if its corresponding partition graph $G_k(P)$ is *acyclic*. For instance, there is no feasible task implementation based on the event partition P_e in Fig. 1(b). In this case, transitions $\{\theta_1, \theta_2, \theta_3, \theta_4\}$ form a *4-cycle*, where a task partition based on events e_1 and e_2 is infeasible. As the smallest possible cycles in event partitions, 4-cycles need to be addressed in any multi-task implementation.

a) Partitioned model: The partitioned model only applies to the FSMs where the event partition graph is acyclic. The multi-task implementation in the partitioned model is based on the event partition. Each task is executed at the period of the corresponding event. The tasks implementing the FSM are activated synchronously with offset $= 0$ and share a variable encoding the current state of the FSM. The task deadlines are assigned to ensure that any transition completes before the next set of inputs is processed. As shown in [8], the absolute deadline of a task instance (implementing transitions/actions) is the minimum value between the end of its period and the earliest activation time of higher priority tasks implementing transitions of the *same FSM*. This deadline guarantees that any task finishes its action execution and state update before a new task instance for the same FSM starts execution, and that there is no preemption between tasks from the same FSM.

b) Mixed-partitioned model: The partitioned model does not apply to the FSMs in which the event partition graph is cyclic. Therefore, a mixed-partitioned model is proposed in [8]. The idea is to identify all the transitions that belong to a cycle in the event partition graph and implement them in a single task τ_b, while the remaining transitions are implemented according to the partitioned model (as in Figure 1(c)). τ_b will be assigned the highest priority and a period equal to the GCD of all the periods of the events that trigger transitions in τ_b.

c) General partitioned model: The mixed-partitioned model proposed in [8] generates a particular task implementation, but many other multi-task implementations are possible, as long as the corresponding task partition graph is acyclic. As shown later in the motivating example, those multi-task implementations may provide better breakdown factor and action extensibility than the mixed-partitioned model. The *general partitioned model* is formally defined as follows.

Definition 1: Given an synchronous FSM F, a task implementation based on the *general partitioned model* is any $\mathcal{T}_F = \{\tau_1, \tau_2, \dots \tau_n\}$ that satisfies the following conditions: (1) any transition in F should be implemented by one and only one task in $\mathcal{T}_F$, i.e. $\forall \theta_i \in F, \exists! \tau_k \in \mathcal{T}_F, s.t.\ \mu(\theta_i) = \tau_k$, (2) any task in $\mathcal{T}_F$ implements at least one transition in F, i.e. $\forall \tau_k \in \mathcal{T}_F, \exists \theta_i \in F, s.t.\ \mu(\theta_i) = \tau_k$, (3) the priorities of tasks in $\mathcal{T}_F$ satisfy all the transition evaluation orders specified in F, i.e. $\forall \theta_i \in F, \theta_j \in F$, if $\theta_i > \theta_j$, then $\pi_{\mu(\theta_i)} > \pi_{\mu(\theta_j)}$.

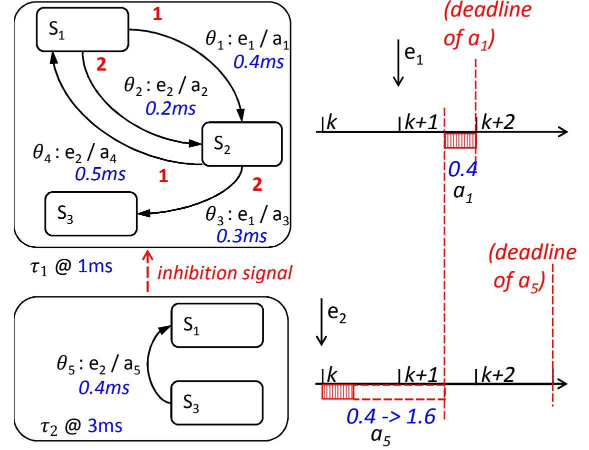

Fig. 2. Extensibility for the third multi-task Implementation

The partitioned model and the mixed-partitioned model are clearly special cases of the general partitioned model.

C. Motivating example

We use the FSM in Fig. 1 as an example to compute the breakdown factor and the action extensibility.

1) Single-task implementation: A single task τ_k implements all transitions. The task period ψ_k is the GCD of event periods $2ms$ and $3ms$, i.e. $1ms$. All transitions must complete within the task period, therefore the breakdown factor is $\psi_k / \max(C_{a_j}) = 1/0.5 = 2$. The extensibility of action a_1 is $\psi_k / C_{a_1} = 1/0.4 = 2.5$. Similarly, the extensibility of a_2 to a_5 are 5, 3.33, 2 and 2.5, respectively.

2) Mixed-partitioned model: As explained earlier, $\{\theta_1, \theta_2, \theta_3, \theta_4\}$ form a 4-cycle that cannot be partitioned based on events. In the mixed-partitioned model, the four transitions are implemented in a single task τ_1 with highest priority and period $1ms$, while θ_5 is implemented in τ_2 with lower priority and period $3ms$. The deadlines of transitions θ_1 to θ_4 are $1ms$; the deadline of θ_5 is also $1ms$ because τ_2 has to finish before the activation of the higher priority task τ_1. In this case, the breakdown factor and the action extensibility are the same as in the single-task implementation.

3) General partitioned model: A simple alternative is to swap the priorities of τ_1 and τ_2, resulting in the implementation of Fig. 2. The periods of τ_1 and τ_2 are still $1ms$ and $3ms$. However, the deadline of θ_5 is now $3ms$ since τ_2 has higher priority, which increases its extensibility. As shown in Fig. 2, assuming the execution of a_5 starts at time k (after θ_5 is triggered by e_2), event e_1 may arrive at time $k+1$ and enable θ_1, with a deadline at $k+2$. To ensure θ_1 completes before $k+2$, the execution time of a_5 can be at most $k + 2 - C_{a_1} - k = 1.6$ (θ_1 is not activated

until θ_5 completes because τ_1 has lower priority, therefore the functional correctness is preserved). It can be proved that this is the worst-case scenario for extending the execution time of a_5, and the extensibility of a_5 is $1.6/0.4 = 4$, larger than in the previous cases. The breakdown factor is still $1/0.5 = 2$.

The last task model improves the action extensibility but not the breakdown factor. A different general partition further improves on both metrics. We implement $\{\theta_1, \theta_2, \theta_3\}$ in τ_1, and $\{\theta_4, \theta_5\}$ in τ_2. τ_2 is assigned a priority higher than τ_1, and a period of $3ms$. τ_1 has a period of $1ms$. The extensibility of a_1, a_2, a_3 is still the same. The extensibility of a_5 is 4, the same as in the last implementation. The extensibility of a_4 increases from $1/0.5 = 2$ to $1.6/0.5 = 3.2$. This is computed by considering the case in which e_1 arrives $1ms$ after θ_4 starts (the analysis is similar as for a_5). Furthermore, the breakdown factor is also reached in this case – e_1 arrives $1ms$ after and both θ_4 and θ_1 need to complete within $2ms$. The breakdown factor is therefore $2/(0.5 + 0.4) = 2.22$ (all actions are scaled by the same factor). It can be proved that the system is schedulable with all tasks scaled by 2.22.

III. Breakdown Factor and Action Extensibility

In this section, we propose methods to compute the breakdown factor and the action extensibility of any given task implementation based on the general partitioned model, for a set of synchronous FSMs implemented on a single processor.

A. Breakdown factor

Definition 2: Given a set of synchronous FSMs $\mathcal{F} = \{F_1, \ldots, F_m\}$ and their task implementation $\mathcal{T}_{\mathcal{F}} = \{\mathcal{T}_{F_1}, \ldots, \mathcal{T}_{F_m}\}$, where each $\mathcal{T}_{F_k}$ is a task implementation of F_k based on the general partitioned model (there may be multiple tasks in $\mathcal{T}_{F_k}$). Let $\mathcal{F}^\lambda$ denote a modification of $\mathcal{F}$ by scaling the execution time of all actions by λ (i.e. $C'_{a_i} = C_{a_i} \cdot \lambda, \forall F_k \in \mathcal{F}, a_i \in F_k$). Let $\mathcal{T}_{\mathcal{F}^\lambda}$ be the task implementation of $\mathcal{F}^\lambda$ with the same transition-to-task mapping and priority assignment as $\mathcal{T}_{\mathcal{F}}$. The breakdown factor $\lambda^{max}_{\mathcal{T}_{\mathcal{F}}}$ of $\mathcal{T}_{\mathcal{F}}$ is the largest λ that can make $\mathcal{T}_{\mathcal{F}^\lambda}$ schedulable. △

To compute the breakdown factor, we use a binary search to explore the range of λ. For each choice of λ, we check the schedulability of $\mathcal{T}_{\mathcal{F}^\lambda}$ based on the *request bound function* (rbf) and *demand bound function* (dbf) for each task $\tau_i \in \mathcal{T}_{F_k{}^\lambda}$. The concepts of rbf and dbf are first introduced in [2] for the analysis of task graphs. In [16], algorithmic solutions are proposed to compute the rbf and dbf for synchronous FSMs, along with the condition for schedulability. Specifically, the rbf of a task τ_i during a time interval $\Delta = [s, f)$, denoted by $\tau_i.rbf(\Delta)$, is the maximum sum of the execution times by the actions that are implemented in τ_i and have their activation time within Δ. The dbf of τ_i during $\Delta = [s, f)$, denoted by $\tau_i.dbf(\Delta)$, is the maximum sum of the execution times by the actions that are implemented in τ_i and have their activation time *and deadline* within Δ. The schedulability of τ_i can be checked by the following condition, where π_i is the priority of τ_i ([16]):

Theorem 3.1: Task τ_i is schedulable if the following constraint is satisfied for all *level-*π_i *busy periods* $[s, f)$:
$\forall t \in [s, f],\ \exists t' \in [s, t]$ such that

$$\tau_i.dbf[s, t] + \sum_{\tau_j \in HP_{\tau_i}} \tau_j.rbf[s, t') \leq t' - s \qquad (1)$$

where HP_{τ_i} consists of tasks with higher priorities than τ_i.

The concept of *level-*π_i *busy period* is introduced in [13]. In a system with periodic trigger events, level-π_i busy periods start at the arrival points of the events within the hyperperiod that trigger tasks with priority higher than or equal to π_i. For instance, assuming there are two events with periods {2, 3}, the start times of the busy periods to be considered are {0, 2, 3, 4} – these are the set of s to be checked in (1). For the choices of t and t' to be checked in (1), we only need to consider the cases where t is the deadline of an instance of τ_i and is within the length of the busy period (for which a bound is derived in [16]) and t' is at the arrival time of an event.

Leveraging Theorem 3.1, we propose Algorithm 1 for computing the breakdown factor of $\mathcal{T}_{\mathcal{F}}$. d_{θ_i} is the shortest deadline for a transition θ_i among all its activations. Let τ_{θ_i} denote the task that implements θ_i (if $\theta_i \in F_k$, then $\tau_{\theta_i} \in \mathcal{T}_{F_k}$). For $\tau_i \in \mathcal{T}_{F_k}$, let $HP^f_{\tau_i}$ denote the tasks that are in $\mathcal{T}_{F_k}$ (i.e. also implementing the transitions from F_k) and have higher priority than τ_i. Θ is the set of all transitions in $\mathcal{F}$, i.e. $\Theta = \{\theta_i \mid \forall F_k \in \mathcal{F}, \theta_i \in F_k\}$.

Algorithm 1 $\lambda^{max}_{\mathcal{T}_{\mathcal{F}}}$ = compute_breakdown_factor($\mathcal{F}$, $\mathcal{T}_{\mathcal{F}}$)

1: $\forall \theta_i \in \Theta$, compute deadline d_{θ_i} as the GCD of the task periods from task set $\{\tau_{\theta_i}\} \cup HP^f_{\tau_{\theta_i}}$
2: $\lambda_{lb} = 0$; $\lambda_{ub} = \min\{d_{\theta_i} \mid \theta_i \in \Theta\}$
3: **while** ($\lambda_{ub} - \lambda lb > MAX_ERROR$) **do**
4: $\quad\lambda = (\lambda_{ub} + \lambda_{lb})/2$
5: $\quad$get $\mathcal{F}^\lambda$ and $\mathcal{T}_{\mathcal{F}^\lambda}$ by scaling C_{a_i} by λ for all action a_i
6: $\quad$schedulable = **true**
7: $\quad$**for all** τ_i in the system **do**
8: $\quad\quad$calculate start points s for level-π_i busy periods
9: $\quad\quad$calculate bound f and choices of t, t' for each busy period
10: $\quad\quad$**for all** (s, t) pair for τ_i **do**
11: $\quad\quad\quad$compute $\tau_i.dbf[s, t]$
12: $\quad\quad\quad$for all t', compute $\tau_j.rbf[s, t')$ and check Constraint 1
13: $\quad\quad\quad$**if** Constraint 1 is not satisfied for any t' **then**
14: $\quad\quad\quad\quad$schedulable = **false**; break out of the loops
15: $\quad$**if** schedulable **then** $\lambda_{lb} = \lambda$ **else** $\lambda_{ub} = \lambda$
16: **return** λ

When computing the rbf and dbf of each task τ_i, we need to consider the state update and take into account the fact that other tasks may be generated from the same FSM F_k. Hence, we derive an FSM F'_k from F_k by setting the computation time of those actions that are not in τ_i to 0, and then compute the rbf and dbf of F'_k. This method is also applied when computing the sum of rbfs and dbfs from several tasks generated from the same FSM.

B. Action extensibility

Action extensibility can be similarly defined as a breakdown factor, with a key difference: the execution time of a single action is increased when computing extensibility.

Definition 3: Given a set of synchronous FSMs $\mathcal{F} = \{F_1, F_2, \ldots F_m\}$ and its task implementation $\mathcal{T}_\mathcal{F} = \{\mathcal{T}_{F_1}, \mathcal{T}_{F_2}, \ldots \mathcal{T}_{F_m}\}$, the action extensibility β_{a_i} of $a_i \in F_k$ is the largest scaling factor that, applied to C_{a_i} (i.e. $C'_{a_i} = C_{a_i} \cdot \beta_{a_i}$) retains the schedulability of the task set. △

Algorithm 2 compute_action_extensibility_single(F, $\mathcal{T}_F$)

1: $\forall \theta_i \in F$, compute deadline d_{θ_i} as the GCD of the task periods from task set $\{\tau_{\theta_i}\} \cup HP_{\tau_{\theta_i}}$
2: **for all** $\theta_i \in F$ **do**
3: $slack^{min} = d_{\theta_i} - C_{a_i}$
4: **for all** θ_j *s.t.* $\tau_{\theta_j} \in LP_{\tau_{\theta_i}}$ **do**
5: $slack = d_{\theta_j} + GCD(\pi_{\theta_i}, \pi_{\theta_j}) - C_{a_i} - C_{a_j}$
6: **for all** $\tau_k \in (HP_{\tau_{\theta_j}} \setminus HP_{\tau_{\theta_i}} \setminus \{\tau_{\theta_i}\})$ **do**
7: $slack = slack - \max\{C_{a_m} \,|\mu(\theta_m) = \tau_k\}$
8: $slack^{min} = \min(slack^{min}, slack)$
9: $\beta_{a_i} = (slack^{min} + C_{a_i})/C_{a_i}$

A similar approach to the breakdown factor can be used for computing the action extensibility, i.e. a binary search on the value of β_{a_i}, checking the system schedulability for each candidate value. However, the rbf and dbf computation for schedulability analysis is very time consuming. If there is only one FSM F on the processor and the largest action execution time is not larger than the GCD of the event periods, a more efficient algorithm (Algorithm 2) can be used to analyze the worst case scenario for action extensibility and directly compute its value with conservative approximations (not overly pessimistic in our tests). The main idea is to check the schedulability of the first instance of any lower priority task after the triggering of a_i. The motivating example shown in Fig. 2 shows the simplest case of such analysis.

IV. Task Implementation Optimization

Several algorithms can be designed to explore different task implementations with respect to the breakdown factor and the action extensibility. The brute force solution is to explore all possible task implementations based on the general partitioned model, and calculate the two metrics for each implementation using Algorithm 1 and 2. However, the complexity is likely to be too high for practical systems. We propose an efficient heuristic that combines local optimization with global correctness checking and can be easily customized to optimize either of the two metrics (the use of different metrics may lead to different optimal solutions).

First, to optimize the action extensibility, we need to extend the extensibility definition to a system level metric.

Definition 4: Given a set of synchronous FSMs $\mathcal{F} = \{F_1, F_2, \ldots F_m\}$ and its task implementation $\mathcal{T}_\mathcal{F} = \{\mathcal{T}_{F_1}, \mathcal{T}_{F_2}, \ldots \mathcal{T}_{F_m}\}$ on a single processor, the *system action extensibility* $B_{\mathcal{T}_\mathcal{F}}$ is defined as the weighted average of the action extensibility of each action a_i in the system, i.e. $B_{\mathcal{T}_\mathcal{F}} = \sum_{a_i \in \bigcup_k F_k} w_i \cdot \beta_{a_i}$ where w_i is a pre-assigned weight that indicates the importance of the action and how likely it will be increased in future upgrades. △

Other options include considering functional dependencies among actions and take into account the simultaneous update of a set of actions (as proposed in [17] for tasks) or to consider the minimum extensibility among all actions in the system. Our algorithm can be easily extended to handle other formulations like those.

The proposed heuristic is shown as Algorithm 3. The algorithm starts by selecting a task implementation for all the transitions forming 4-cycles in the event partition graph. For each set of transitions in a 4-cycle, all the feasible task implementations are explored by the function *optimal_task_implementation*, which returns the best local solution for the breakdown factor or the action extensibility (according to the selected metric, ties are broken in favor of the mapping with less tasks). Next, all remaining transitions are temporarily mapped to a dedicated task. Then, all cycles in the resulting task graph are removed by merging the corresponding tasks (generating a feasible task partition). Next, opportunities for merging pairs of tasks activated by the same event are explored (the function *if_mergeable* checks whether merging two tasks will result in a cycle in the task graph). The result of this stage depends on how the task pairs are selected for trying a merger – in our algorithm, they are selected in order of their weights and execution times. Finally, priorities are assigned to tasks in agreement with the transition evaluation order. If no order is defined between any two task partitions, then their priorities are assigned using a reverse rate monotonic rule (to maximize task deadlines).

Algorithm 3 optimize_task_implementation($\mathcal{F}$)

1: **for all** $F_k \in \mathcal{F}$; $\Theta_k = \{\theta_i \mid \theta_i \in F_k\}$ **do**
2: $\mathcal{T}_{\mathcal{F}_k} = \emptyset$
3: **while** $\exists$ a 4-cycle $\mathcal{Q} = \{\theta_1, \theta_2, \theta_3, \theta_4\} \in \Theta_k$ **do**
4: $\mathcal{T}'$ = optimal_task_implementation($\mathcal{Q}$)
5: $\mathcal{T}_{\mathcal{F}_k} = \mathcal{T}_{\mathcal{F}_k} \cup \mathcal{T}'$; $\Theta_k = \Theta_k \setminus \mathcal{Q}$
6: **for all** $\theta_i \in \Theta_k$ **do**
7: generate one task τ_i for each θ_i; $\mathcal{T}_{\mathcal{F}_k} = \mathcal{T}_{\mathcal{F}_k} \cup \{\tau_i\}$
8: Build partition graph $G_k(\mathcal{T}_{\mathcal{F}_k})$
9: **while** $\exists$ a cycle $(\tau_1, \tau_2, ..., \tau_n)$ in $G_k(\mathcal{T}_{\mathcal{F}_k})$ **do**
10: merge tasks $\tau_1, \tau_2, \ldots, \tau_n$ into τ_m
11: update $\mathcal{T}_{\mathcal{F}_k}$ and $G_k(\mathcal{T}_{\mathcal{F}_k})$
12: **while** $\exists$ $\tau_i, \tau_j \in \mathcal{T}_{\mathcal{F}_k}$ *s.t.* both τ_i and τ_j are triggered by the same event **and** if_mergeable(τ_i, τ_j) **do**
13: merge τ_i and τ_j to task τ_m
14: $\mathcal{T}_{\mathcal{F}_k} = \mathcal{T}_{\mathcal{F}_k} \cup \{\tau_m\} \setminus \{\tau_i, \tau_j\}$, and update $G_k(\mathcal{T}_{\mathcal{F}_k})$
15: Assign task priorities based on transition priorities and task periods

V. Experimental Results

We use the TGFF tool [9] to generate random graphs and extend them to random FSM models (with added transitions, randomly selected periods, execution times and priorities) as our experiment sets. We compare the task implementations generated from our Algorithm 3 with the single-task implementation of the FSMs.

First, we compare the system action extensibility, for cases with a single FSM having from 5 to 25 states. The results are shown in Fig 3. The line labeled *harmonic-fixed* represents the cases where all event periods are harmonic and the transition priorities entirely depend on their events (the event partition is feasible). The line *harmonic-50%* represents the cases

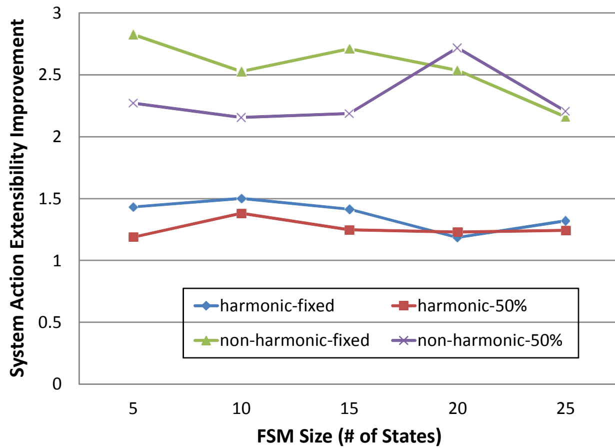

Fig. 3. System Action Extensibility Improvement

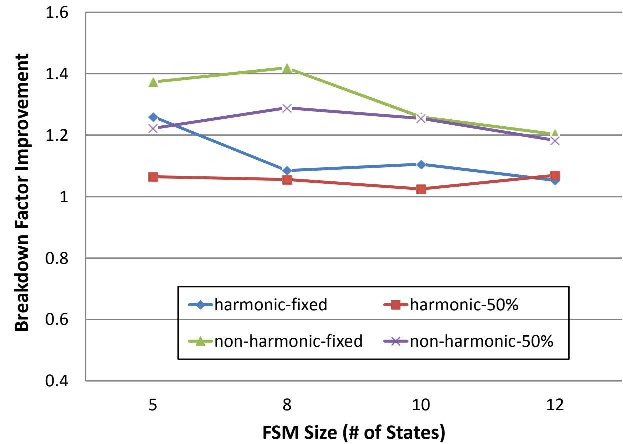

Fig. 4. Breakdown Factor Improvement

with harmonic periods and the transition evaluation orders depending on their events with a 50% probability (which might lead to 4-cycles). The other two lines are for non-harmonic event periods. Each data point in the figure is the average result of 20 random FSMs. For harmonic cases, our algorithm provides an improvement between 20% to 50% over the single-task implementation. For non-harmonic sets, the improvement increases to between $2X$ and $3X$. Non-harmonic sets allow more improvement since the deadlines in our multi-task implementations are typically much larger than in the single-task solution. When computing the action extensibility of our task implementation, we use Algorithm 2 since this is a single-FSM system. Action extensibility can be computed within 10 minutes for FSMs with 250 states.

Next, we compare the breakdown factor. Because of the timing complexity for computing the breakdown factor in Algorithm 1, we tested FSMs with size up to 12 states. The results are shown in Fig. 4, for the same sets of randomly generated FSMs used in the extensibility case. For non-harmonic cases, the improvement is between 20% and 40%, while for harmonic cases it is around 10%. The improvement on the breakdown factor is less than the action extensibility, because it is limited by the action with the smallest timing slack, which is harder to improve.

Finally, we compare the system action extensibility and breakdown factor, for systems where multiple FSMs are implemented on a processor. In both cases, we need to use rbf and dbf for schedulability analysis, and therefore we were only able to test systems that have 3 FSMs and each FSM has 5 states. The average improvement on system action extensibility is 70% and the improvement on the breakdown factor is 30% (both for non-harmonic and fixed priority cases).

VI. Conclusion

We define a general partitioned model for multi-task implementations of synchronous FSMs and two metrics for measuring the quality of task implementations: the breakdown factor and the action extensibility. We propose an algorithm to explore robust and extensible task implementations based on the two metrics. In future work, we plan to improve the method for computing the two metrics by developing approximated schedulability tests. We also plan to consider the timing and memory overhead of the task implementations.

References

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

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

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

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

[5] M. Di Natale and V. Pappalardo. Buffer optimization in multitask implementations of simulink models. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–32, 2008.

[6] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. Synthesis of Multi-task Implementations of Simulink Models with Minimum Delays. *IEEE Trans. Industrial Informatics*, 6(4):637–651, 2010.

[7] S. Tripakis, C. Sofronis, N. Scaife, and P. Caspi. Semantics-preserving and memory-efficient implementation of inter-task communication on static-priority or edf schedulers. Proc. of the 5th ACM EMSOFT conference, 2005.

[8] M. Di Natale and H. Zeng. Task Implementation and Schedulability Analysis of Synchronous Finite State Machines. In *Proc. the Conference on Design, Automation and Test in Europe*, 2012.

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

[10] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Trans. Computer-Aided Design*, 21(2):169–183, Feb. 2002.

[11] E. Lee and P. Varaiya. Structure and Interpretation of Signals and Systems. *Addison Wesley*, 2003.

[12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Real Time Systems Symposium*, 1989.

[13] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. Real-Time Systems Symposium*, 1990.

[14] N. Scaife, C. Sofronis, P. Caspi, S. Tripakis, and F. Maraninchi. Defining and translating a "safe" subset of Simulink/Stateflow into Lustre. *4th ACM International Conference on Embedded Software*, 2004

[15] D. Weil, V. Berlin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of Esterel for real-time embedded systems. *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Syst.*, 2000.

[16] H. Zeng and M. Di Natale. Schedulability analysis of periodic tasks implementing synchronous finite state machines. In *Proc. 24th Euromicro Conference on Real-Time Systems*, 2012.

[17] Q. Zhu, Y. Yang, E. Scholte, M. Di Natale, and A. Sangiovanni-Vincentelli. Optimizing the Software Architecture for Extensibility in Hard Real-Time Distributed Systems. *IEEE Transactions on Industrial Informatics*, 6(4): 621–636, November 2010.