A Model-based Synthesis Flow for Automotive CPS

Peng Deng, Qi Zhu
UC Riverside
pdeng002@ucr.edu, qzhu@ece.ucr.edu

Fabio Cremona, Marco Di Natale
Scuola Superiore S. Anna
{f.cremona, marco}@sssup.it

Haibo Zeng
Virginia Tech
hbzeng@vt.edu

***Abstract*—Synchronous reactive models are used by automotive suppliers to develop functionality delivered as AUTOSAR components to system integrators (OEMs). Integrators must then generate a task implementation from runnables in AUTOSAR components and deploy tasks onto CPU cores, while preserving timing and resource constraints. In this work, we propose an integrated synthesis flow that addresses both sides of the supply chain. On the supplier side, from synchronous models, we generate AUTOSAR runnables that promote reuse and ease the job of finding schedulable implementations. On the integrator side, we find the mapping of runnables onto tasks and allocation of tasks on cores that satisfy the timing constraints and are memory efficient.**

I. Introduction

Model-based design is being increasingly used in the development of cyber-physical systems due to its capability to support early design verification/validation through formal functional models and to generate software implementations from models [**? ? ?**]. In the automotive domain, model-based design and the advent of the AUTOSAR standard [**?**] for the specification of software components are changing the way the automotive electronics toolchain operates from electronics systems suppliers (TIER 1) to system integrators or carmakers.

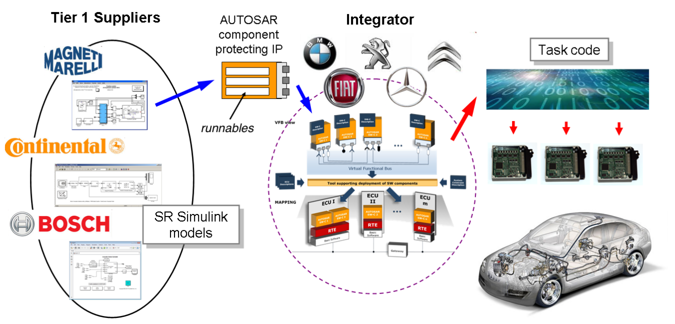

Fig. 1. The automotive software systems supply flow.

As shown in Fig. **??**, many automotive controls are today specified and realized using Synchronous Reactive (SR) models such as those modeled in the Simulink/Stateflow tool. In Simulink, the control functionality is defined (by TIER 1 suppliers) as a network of blocks, organized in a hierarchy of subsystems or higher level components. In the software supply flow, a code implementation is generated for each subsystem as a set of *runnables* (functions), and provided with the black-box functional specification of the component as an AUTOSAR software component or SWC. The code functions are typically provided in the format of object code or library, together with an AUTOSAR model that expresses the data dependencies, the order of execution dependencies, the activation events or the call dependencies for all the functions. Then, the integrators (carmakers or OEMs) collect all the SWCs from the TIER 1 suppliers, connect them in a system-level model, and generate the task configuration by mapping runnables onto tasks and allocating tasks to the CPUs, assisted by AUTOSAR tools.

In this work, we propose a set of synthesis algorithms for *automating and optimizing* the steps in the above software supply flow. For the first step, we generate a set of runnables from synchronous models with multiple objectives: The runnables should try to hide the internal structure of the synchronous models for IP protection, and at the same time should be reusable (without false dependencies) and allow for an easy integration and scheduling. For the second step, we generate task implementations from runnables to achieve a schedulable and resource efficient deployment.

A. State of the Art

A number of formal definitions belong to the general category of SR models, such as the Signal, Lustre, and Esterel [**?**]. Esterel or Lustre models are typically implemented as a single executable that runs according to an event server model. Reactions to events are decomposed into atomic actions that are partially ordered by the causality analysis of the program. The scheduling is generated at compile time and tries to exploit the partial causality order of functions. The generated code executes without the need of an operating system [**? ?**].

The Simulink/Stateflow toolset [**?**] is very popular among control system designers. The Simulink code generator produces a *Step* function for each subsystem block (a subset of a component). When generating code from a model, subsystem blocks are clustered and often provided to the integrator in binary form with a higher level model of the component.

The problem of clustering synchronous blocks in functions (runnables) for modular reuse by avoiding false input-output dependencies is discussed in [**?**], with solutions for single-rate and multirate systems. In [**?**], the trade-off between modularity and code size is further explored. Optimal disjoint clustering is demonstrated to be an NP-complete problem and encoded as a Boolean satisfiability (SAT) problem. In [**?**], an efficient symbolic representation is proposed to simplify the modularity optimization with reusability consideration.

The problem of finding the optimal definition of a multitask implementation for multirate Simulink models scheduled with fixed priority on a single-core is discussed in [**?**] where a branch-and-bound algorithm is presented. In [?], an MILP (Mixed Integer Linear Programming) formulation is provided.

The problem of mapping single-rate SR blocks to functions and tasks with timing constraints and performance objectives is discussed in [?], where the synthesis methods in [?] and [?] are extended to consider execution times and timing constraints. In [?], the Firing Time Automaton formalism is proposed for expressing firing times of multirate components.

The Prelude synchronous language [?] includes rules and operators for the selection of a mapping onto platforms with Earliest Deadline First (EDF) scheduling, including symmetric multicore architectures. Communication among nodes executing at different rates is realized using the protocol and data structures defined in [?] (an extension of [?] which is optimal with respect to the number of required buffers for a system containing only periodic nodes).

The problem of mapping AUTOSAR runnables onto ECUs in a distributed system with the objective of bus load minimization is discussed in [?] and [?], with solutions based on a genetic algorithm. The analysis of real-time communication mechanisms based on spin locks for AUTOSAR runnables in multicores is presented in [?], and mechanisms for a flow-preserving AUTOSAR implementation of synchronous signals are presented and compared in [?]. The optimal placement of tasks with deadline constraints communicating with locking primitives in multicores is discussed in [?] [?], where an MILP solution and effective heuristics are presented.

B. Contributions

Existing research works either consider the automotive software synthesis problem as a one-step process with limited optimization, or treat different aspects of software synthesis as isolated steps and focus on only part of them. For instance, the generation of runnables is often conducted without consideration of timing and schedulability implication, and the mapping from runnables to tasks is often conducted separately from the mapping of tasks to cores. In this work, we provide:

- An integrated model-based software synthesis flow, with formal identification of synthesis steps from synchronous models to AUTOSAR runnables and from runnables to a task implementation on multicores.
- A set of algorithms for optimizing the synthesis steps with respect to multiple design metrics, among which schedulability is being addressed throughout the entire flow. In particular, the problem of generating runnables with schedulability consideration and the problem of mapping runnables with partial execution order and flow preservation constraints on multicores have not been discussed before.
- The formalism of Firing and Execution Time Automaton (FETA) for capturing the worst-case execution time (WCET) of blocks and runnables at each activation. FETA is utilized throughout our synthesis flow for timing request description.

In this work, three system views are considered, the Synchronous (or SR) view, the Runnables (or AUTOSAR) view, and the Task view (or Task model). The SR view and the Runnables view are the input and output of the first synthesis step, while the Runnables view and the Task model are the input and output of the second synthesis step.

II. From SR Models to AUTOSAR Runnables

The first step of the synthesis flow is to synthesize multirate SR models to runnables.

A. System Model

In the **Synchronous view**, a system is a synchronous block diagram (SBD) composed of a set of subsystems or components $\mathbf{S} = \{\mathcal{C}_j\}$. Each component $\mathcal{C}_j$ consists of a network of blocks $B^j = \{b_1^j, b_2^j, \ldots, b_{m^j}^j\}$ connected by links representing functional dependencies, a set of inputs $X^j = \{x_1^j, x_2^j, \ldots, x_{p^j}^j\}$ and a set of outputs $Y^j = \{y_1^j, y_2^j, \ldots, y_{q^j}^j\}$. The output y_k^j depends on the input x_p^j, denoted as $D(y_k^j, x_p^j)$, if there is a path of links and blocks (indicating a causal dependency) between x_p^j and y_k^j. $X(b_k^j)$ and $Y(b_k^j)$ denote the set of input and output ports directly connected to block b_k^j, respectively. Each block b_i^j has an activation period t_i^j and a WCET γ_i^j. Whenever there is an input-output dependency between a sender block b_i^j and a receiver block b_k^j for which the output update function depends on the input values (b_k^j is of type *feedthrough*), there is an execution order constraint between the two blocks (and their implementations) $b_i^j \rightarrow b_k^j$. For every b_i^j, $Pred(b_i^j)$ is the set of its predecessors, and $Succ(b_i^j)$ is the set of its successors. The starting synchronous model may have cycles in the data paths, but there must be no cycles in the set of the execution order constraints. This means that whenever there is a cycle in the model, for at least one of the blocks or subsystems in the cycle the outputs should not depend instantaneously on the inputs, but only on the block state (the block is of Moore type).

The component of Fig. **??** with 10 blocks, labeled A to J, is used as a running example. The block periods are shown in their right side and their WCET on the bottom-left corner. Functional dependencies between blocks are shown as links.

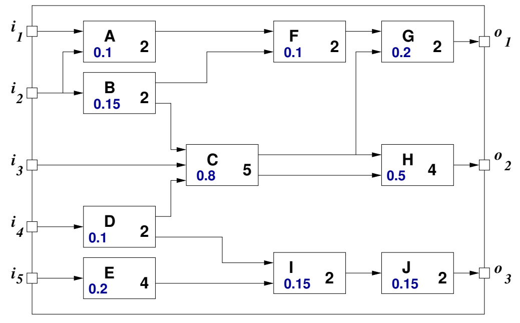

Fig. 2. Illustrating example

In the **Runnables view**, an implementation $\mathcal{R}_j$ consisting of a set of runnables $\mathcal{R}_j = \{r_1^j, r_2^j \ldots r_p^j\}$ is provided for each component $\mathcal{C}_j$. Each runnable is the code implementation of a subset of the blocks in $\mathcal{C}_j$, denoted as $BI(r_k^j) = \{b_{k1}^j, b_{k2}^j, \ldots b_{ks}^j\}$ with the requirement that the union of all the runnable blocksets covers all the blocks in $\mathcal{C}_j$. Some blocks may be allocated to more than one runnables, in which case a mechanism in the generated code ensures that the block implemented by multiple runnables is only executed once by the runnable that executes first. Each runnable r_k^j has a base period ϕ_k^j corresponding to the greatest common divisor of all the blocks in $BI(r_k^j)$. A set of partial execution order constraints $r_k^i \to r_k^j$ is defined on runnables. Each runnable will read from (write into) the input (output) ports accessed by the blocks mapped onto it. By extending the previous notations, $X(r_k^j)$ and $Y(r_k^j)$ denote the set of input and output ports accessed by runnable r_k^j with $X(r_k^j) = \bigcup_t X(b_{kt}^j)$.

B. Using FETA for the Specification of Activations and Execution Time Requests

The runnables implementation of a component must be a correct implementation of the synchronous model and provides the input for the task synthesis. The task model must guarantee that all reactions complete before the next activation event. To verify this property, the WCETs of the runnables and the blocks mapped onto them need to be properly expressed. The timing specification needs to represent the time instants at which the execution of the runnable is requested and the amount of processing time that is requested for each activation.

A runnable is the implementation of a set of blocks (possibly with different periods) for which execution is requested periodically. A Firing Time Automaton (FTA) [**?**] is a formalism to describe activation events for components that result from the union of synchronously-activated periodic systems. We provide an adapted description here (and also simplified – the original definition allows for activation offsets). An FTA $\mathcal{A}$ is a pair $\mathcal{A} = (\theta, S)$ where θ is a (base) period and $S = (V, v_0, F, \delta)$, where $V = \{v_0, v_1, \ldots v_n\}$ is a set of vertices/states, $v_0 \in V$ is the initial vertex, $F \subseteq V$ is the subset of firing (double lines in the figures) vertices, indicating a point in time when one or more blocks are triggered, and $\delta : v_{i-1} \to v_i$ is a link or transition, with $v_n \to v_0 \in \delta$, i.e. the FTA is a cycle. The cycle period of the FTA is $\Theta = \theta * n$.

Extending FTA: The FTA description is not sufficient for our purpose, and needs to be extended with the representation of the requested execution time at each activation instant. In addition, to account for the possibility that a block is mapped onto multiple runnables, execution time needs to be expressed in a conditional way, depending on the runnable execution order. In our FETA extension, each firing vertex v_p is associated with a WCET specification list of the type $W_p = \{(c_{p0}, -), (c_{pk}, r_k)*\}$, where the first WCET is unconditional, and the other WCETs in the list (with $c_{pk} < c_{p0}$) are conditional to the previous execution of the runnable r_k for the same activation event (in order to account for blocks mapped onto multiple runnables, the component index is dropped because the definition only applies to runnables from the same component). For convenience, CE_p denotes the set of all the runnables indexes k that appear in the conditional execution list of W_p. A sample runnable generation of our example, with the execution time requirements indicated with an FETA notation is in the Fig. **??**. In the figure, the data paths from the component ports to the internal runnables are shown as solid lines, the execution order constraints between runnables (derived by those between the blocks inside them) are shown as dashed lines.

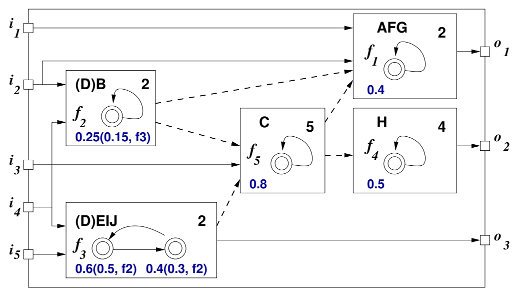

Fig. 3.   A sample runnable generation of the Fig. **??** example

The FETA specification for a runnable can be easily constructed from the timing specifications (t, γ) of its blocks. Also, an FETA specification can be computed when merging runnables into tasks. In this case, operators are needed to compute the merger. Two FETAs are equivalent if their firing times occur at exactly the same time instants and for the same (conditional) specification of execution time.

Operators and Composition: Two operators are required for the composition of FETA specifications: an oversampling operator, which transforms an FETA $\mathcal{A} = (\theta, S)$ into an equivalent $\mathcal{A}' = (\theta', S')$ with $\theta = k \times \theta'$ for some integer k, and a sum operator $\mathcal{A} = \mathcal{B} \oplus \mathcal{C}$ with $\theta_B = \theta_C$ (defined for two FETAs with the same base period).

For the **oversampling** operator, the structure of $\mathcal{A}'$ is $S' = (V', v_0', F', \delta')$, with $\|V\| = k * n$. For each $v_i \in V$, with $i = 0, \ldots, n$, there is a set of k vertices $\{v'_{i*k}, v'_{i*k+1}, \ldots v'_{i*k+k-1}\} \in V'$, where $v'_{i*k} \in F'$ if $v_i \in F'$ (and $W'_{i*k} = W_i$), and $v'_{i*k+j} \notin F'$ for all $j = 1, \ldots, k-1$. Finally $v'_j \to v'_{j+1} \in \delta'$ for all $j = 0, \ldots, k*n-1$ and $v'_{k*n} \to v'_0 \in \delta'$. The result of the **sum operation** $\mathcal{A} = \mathcal{B} \oplus \mathcal{C}$ is $S^A = (V^A, v_0^A, F^A, \delta^A)$ with $\|V^A\| = n^A = lcm(n^b, n^C)$ and $v_j^A \to v_{j+1}^A \in \delta^A$ for all $j = 0, \ldots, n^A - 1$ and $v_{n^A}^A \to v_0^A \in \delta^A$. For each $v_i^A \in V^A$, there are two nodes $v_j^B \in \mathcal{B}$ and $v_k^C \in \mathcal{C}$ such that $j = i \mod n^B$ and $k = i \mod n^C$ (where $\mod$ is the modulo operator).

Definition of firing nodes: $v_i^A \in F^A$ if $v_j^B \in F^B \vee v_k^C \in F^C$.

Definition of execution time for a firing node: If $v_j^B \in F^B$ and $v_k^C \notin F^C$, then $W_i^A = W_i^B$. Conversely, if $v_j^B \notin F^B$ and $v_k^C \in F^C$, then $W_i^A = W_i^C$. If $v_j^B \in F^B \wedge v_k^C \in F^C$, then $CE_A = CE_B \cup CE_C$. For *unconditional executions*, it is (assuming the runnable index matches the FETA denomination)

$$c_{A0} = \begin{cases} c_{B0} + c_{C0} & \text{if } B \notin CE_C \wedge C \notin CE_B \\ c_{B0} + c_{CB} & \text{if } B \in CE_C \\ c_{BC} + c_{C0} & \text{if } C \in CE_B \end{cases}$$

Similarly, $\forall k \in CE_A$ (which implies $k \neq B, C$).

$$c_{Ak} = \begin{cases} c_{Bk} + c_{Ck} & \text{if } k \in CE_C \wedge k \in CE_B \\ c_{B0} + c_{Ck} & \text{if } k \notin CE_B \wedge k \in CE_C \\ c_{Bk} + c_{C0} & \text{if } k \in CE_B \wedge k \notin CE_C \end{cases}$$

C. Objectives and Constraints

Each runnable inherits inputs, outputs, and execution order constraints from the blocks mapped onto it. Runnable synthesis, i.e. the **mapping of blocks to runnables**, is subject to the constraints of execution order and activation rate, and evaluated on the metrics of modularity, reusability and schedulability. The execution order constraint requires that all blocks are implemented by runnables (calling their update functions) according to their execution order. The rate constraints require that each block is mapped to a runnable whose period is an integer divisor of the block's period. The three metrics for evaluating runnable generation are introduced in below.

Modularity (IP disclosure): A runnable generation hides information of the internal block structure if the number of runnables (and their dependencies) is significantly smaller than the number of internal blocks. Modularity is measured by the number of runnables generated.

Reusability: A runnable generation is reusable if it does not introduce any false input-output dependencies.

The concept of reusability is defined and investigated in [**? ? ? ?**]. Fig. **??** shows a simple example. Diagram S includes three blocks, two inputs and two outputs. If S is implemented as a single runnable (with a single-function code implementation of S such as `o1, o2 = S.step(i1, i2)`) and reused as a black-box, the feedback connection on the top right of Fig. **??** is not allowed, leading to limitations in its reuse. In reality, there is no dependency between o_1 and i_2 and the feedback composition in the top-right is safe since it does not result in any functional loop.

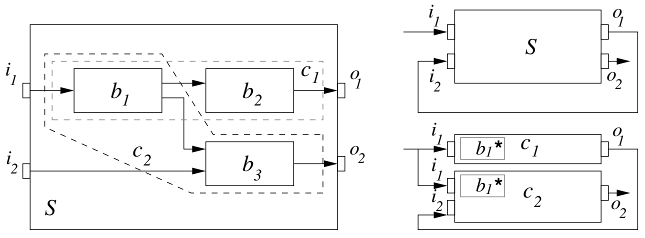

Fig. 4. Different runnable generations result in different reusability

To improve reusability, different runnable generations could be explored, such as using two runnables: c_1, which includes b_1 and b_2 and computes o_1 from i_1; and c_2, which includes b_1 and b_3 and computes o_2 from i_1 and i_2. The connection from o_1 to i_2 becomes possible even when c_1 and c_2 are considered as black-boxes, because no loop exists, as shown in the bottom right of the figure. b_1 is part of both runnables in this implementation. A binary flag is associated with the invocation of the block update function for b_1. The first runnable executing the block sets the flag so that the other can skip the execution of b_1. Note that this runnable generation introduces overhead on code size.

For the running example in Fig. **??**, if we simply map all blocks into a runnable, we have false dependencies between output o_1 and input i_5, o_2 and i_1, etc. In comparison, the runnable implementation in Fig. **??** does not introduce false dependencies and therefore provides maximum reusability. However the modularity is worse (more runnables).

Schedulability: A runnable implementation eases schedulability if it does not introduce scheduling bottlenecks. A quantitative metric expressing this requirements can be computed as follows. Any firing vertex v_i of any FETA $\mathcal{F}_k$ is further characterized by its local utilization $u_i = c_{i0}/\rho_i$, where $\rho_i = \theta_k * d_i$ and d_i is the length of the FETA path from v_i to the next firing vertex (c_{i0} is the unconditional WCET at v_i and θ_k is the period of $\mathcal{F}_k$ as defined earlier).

For a component $\mathcal{C}$ (the index is dropped for simplicity) and one of its runnable implementation $\mathcal{R}$, the local utilization of vertex v in the k^{th} runnable r_k is denoted as u_k^v. The sum of the maximum local utilizations for $\mathcal{R}$ is denoted as U_l and can be computed as:

$$U_l = \sum_{r_k \in \mathcal{R}} u_k^{\max} \quad \text{where } u_k^{\max} = \max_v(u_k^v) \tag{1}$$

The component utilization U_c is computed as:

$$U_c = \sum_{b_i \in \mathcal{C}} \frac{C_i}{T_i} \tag{2}$$

where C_i and T_i are the computation time and period of the block i in the component.

We define the ***alpha ratio*** $\alpha_u = U_l/U_c$ as the ratio between the sum of maximum local utilizations and the component utilization. α_u is used to estimate the schedulability of the runnable implementation (smaller α_u indicates better schedulability). The following theorem helps identify the runnable generation with maximum schedulability.

Theorem 2.1: The alpha ratio $\alpha_u = \frac{U_l}{U_c}$ for any runnable implementation $\mathcal{R}$ is greater than or equal to 1, i.e. $U_l \geq U_c$. The ratio is 1 if and only if the blocks with the same period are grouped in the same runnable.

Proof: For the FETA $\mathcal{F}_k$ of a runnable, let d' be the path length from the vertex at which all the blocks in $\mathcal{F}_k$ are triggered to the next firing vertex. Let $C_{k,j}$ and $T_{k,j}$ denote the computation time and period of block j in the runnable r_k. Then, we have:

$$U_l = \sum_k \max_v(u_k^v) \geq \sum_k \frac{\sum_j C_{k,j}}{d'}$$
$$\geq \sum_k \frac{\sum_j C_{k,j}}{\min_j\{T_{k,j}\}} \geq \sum_k \sum_j \frac{C_{k,j}}{T_{k,j}} = U_c$$

It is easy to see that $U_l = U_c$ if and only if $\min_j\{T_{k,j}\} = T_{k,j}$ for any j, i.e. all the blocks in r_k have the same period. ■

Intuitively, schedulability is maximized when $\alpha_u = 1$.

D. Runnable Synthesis Algorithms

Utilizing the above definitions of FETA and alpha ratio, we propose a top-down method and a bottom-up method for runnable synthesis, to trade off schedulability and modularity while achieving maximum reusability and minimum code size (each block is mapped to one runnable).

Top-down Method:

In the top-down method, we first use an MILP formulation to find a runnable generation with optimal modularity, i.e. the minimal number of runnables, while achieving the maximum reusability. Next, a heuristic gradually decomposes the runnables to improve schedulability, until the desired alpha ratio is obtained. Maximum reusability is preserved during the process, since the decompositions in the heuristic part will not introduce any false input-output dependencies.

The initial *MILP formulation* for obtaining optimal modularity under maximum reusability is as follows (similar as in [**?**]). Maximum reusability can be expressed using constraints that refer to the causal relationships among outputs and inputs. Each input or output is represented by a virtual block. X and Y denote the set of input and output blocks, respectively, and $\mathcal{B}$ denotes the original block set. An input or output (pseudo) block can only be mapped to the same runnable as the original block reading/driving the inputs/outputs. C_{b_i} and C_{r_m} denote the component to which b_i and r_m belong. The binary variable g_{b_i,r_m} represents whether the block b_i is mapped to runnable r_m, and a_{b_i,b_j} denotes whether b_i and b_i are mapped to the same runnable. The variable h_{r_m,r_n} captures the causality relation between runnables (if 1 then r_m must be executed before r_n) and the parameter Q_{b_i,b_j} records the causality relation between blocks and can be computed by using a BFS (Breath-first search) before running the MILP optimization. Z_{r_m} indicates whether there is at least a block mapped to runnable r_m, and therefore $\sum_m Z_{r_m}$ represents the number of runnables. The constraints on block-to-runnable mapping are (**??**–**??**). Constraints (**??**–**??**) define the causality relations among tasks, and (**??**) requires that the block-to-runnable mapping does not introduce any false input-output dependencies.

$$\min \sum_m Z_{r_m} \tag{3}$$

$$Z_{r_m} \geq g_{b_i,r_m} \quad \forall b_i, r_m \in \mathcal{B}, \mathcal{R} \tag{4}$$

$$\sum_m g_{b_i,r_m} = 1 \quad \forall b_i, r_m \in \mathcal{B}, \mathcal{R} \tag{5}$$

$$g_{b_i,r_m} = 0 \quad \forall C_{b_i} \neq C_{r_m} \tag{6}$$

$$g_{b_i,r_m} = 1 \quad \forall b_i, r_m \in X, Y \wedge m = i \tag{7}$$

$$g_{b_i,r_m} = 0 \quad \forall b_i, r_m \in X, Y \wedge m \neq i \tag{8}$$

$$h_{r_m,r_n} \geq g_{b_i,r_m} + g_{b_j,r_n} + Q_{b_i,b_j} - 2 \tag{9}$$

$$h_{r_m,r_n} \geq h_{r_m,r_l} + h_{r_l,r_n} - 1 \tag{10}$$

$$h_{r_m,r_n} + h_{r_n,r_m} \leq 1 \tag{11}$$

$$h_{r_m,r_n} = Q_{b_m,b_n} \quad \forall b_m, r_m \in X \wedge b_n, r_n \in Y \tag{12}$$

Starting from the solution obtained from the MILP formulation, the *heuristic Algorithm* **??** gradually decomposes runnables to improve schedulability. At each step, the algorithm finds the runnable r_m that has the maximum local utilization among those that contain blocks with different periods. It then tries to decompose f_m by moving some of the blocks from f_m to a new runnable f_n without introducing cyclic dependencies (line **??** to **??**). First, it tries to find a block b_i that can be moved to r_n without introducing cycles (blocks are sorted by their potential of reducing α). Then it moves to r_n all the other blocks with the same period that will not add a cycle when added to r_n. The new r_n will only have blocks with the same period, thereby reducing α.

Algorithm 1 Top-down heuristic for runnable synthesis

```
 1: Obtain a runnable generation $p$ with optimal modularity with $N$ runnables using the MILP formulation.
 2: Let $\alpha'_u$ be the target value for $\alpha_u$ (alpha ratio), $N'$ is the desired number of runnables.
 3: $\alpha_u$ = ComputeAlpha()
 4: while $\alpha_u > \alpha'_u \vee N < N'$ do
 5:     Let $r_m$ denote the runnable with maximum $U_l$ among those with different period blocks.
 6:     $\forall b_i \in r_m$, order $b_i$ based on descending $T_{b_i}$, $C_{b_i}$
 7:     $r_n = \Phi$, found = false, $T_n = 0$
 8:     for all $b_i \in r_m$ in order do
 9:         if found = false then
10:             $r_n = r_n \cup b_i$
11:             $p_t$ = decompose($r_m$, $r_n$)
12:             if existsCycle($p_t$) then
13:                 $r_n = r_n - b_i$
14:             else
15:                 $p_b = p_t$, found = true, $T_n = T_{b_i}$
16:         else
17:             if $T_{b_i} = T_n$ then
18:                 $r_n = r_n \cup b_i$
19:                 $p_t$ = decompose($r_m$, $r_n$)
20:                 if existsCycle($p_t$) then
21:                     $r_n = r_n - b_i$
22:                 else
23:                     $p_b = p_t$
24:     $p = p_b$, $\alpha_u$=ComputeAlpha(), $N = N + 1$
25: return $p$
```

Algorithm **??** does not have to start with a schedulable solution (e.g., the max local utilization may be larger than 1), and in rare cases may terminate without finding a feasible solution. In those cases, a pre-processing step can be added to explore more (schedulable) starting points.

Bottom-up Method:

In the bottom-up method, the initial MILP formulation finds a runnable generation with maximum schedulability, i.e. minimum alpha ratio, while achieving the maximum reusability. Then a heuristic gradually merges runnables to improve modularity. During the merging process, input-output dependencies are checked to ensure that maximum reusability is preserved. According to Theorem **??**, schedulability is eased when runnables only contain blocks with the same rate. The bottom-up MILP formulation maximizes the schedulability in runnable synthesis by enforcing such constraint on block periods, while achieving the maximum reusability and the best possible modularity. It is similar to the MILP formulation used in the top-down method, except for the following *additional constraints* (**??**–**??**) on block periods to ensure that all blocks in a runnable have the same period.

$$a_{b_i,b_j} \geq g_{b_i,r_m} + g_{b_j,r_m} - 1, \quad \forall b_i, b_j, r_m \in \mathcal{B} \tag{13}$$

$$a_{b_i,b_j} \leq g_{b_i,r_m} - g_{b_j,r_m} + 1, \quad \forall b_i, b_j, r_m \in \mathcal{B} \tag{14}$$

$$a_{b_i,b_j} \leq g_{b_j,r_m} - g_{b_i,r_m} + 1, \quad \forall b_i, b_j, r_m \in \mathcal{B} \tag{15}$$

$$a_{b_i,b_j} = 0, \quad \forall T_{b_i} \neq T_{b_j} \tag{16}$$

Starting from the MILP solution, the *heuristic Algorithm* **??** explores all possible merges between two runnables, and selects the merge that results in the least increase in the sum of maximum local utilizations, while retaining maximum reusability. The merging process continues until the number of runnables reaches the desired modularity level or the alpha ratio becomes larger than a given threshold, or no schedulable merges can be found.

Algorithm 2 Bottom-up heuristic for runnable synthesis

```
1: Obtain a runnable generation $p$ with optimal schedulability.
2: Let $\Delta U$ = inf and $N'$ denote the desired number of runnables, $\alpha'_u$ is
   the desired $\alpha_u$ value,
3: $U_l$ = compLocalUtil($p$)
4: $\alpha_u = U_l/U_c$
5: while $N > N' \wedge \alpha_u < \alpha'_u$ do
6:     for all $r_m, r_n \in p$ do
7:         $p_t$ = merge($r_m$,$r_n$)
8:         if isSched($p_t$) then
9:             $U_l^{pt}$ = compLocalUtil($p_t$)
10:            if $\Delta U > U_l^{pt} - U_l$ then
11:                $p_b = p_t$, $\Delta U = U_l^{pt} - U_l$
12:    if $\exists p_b$ then
13:        $p = p_b$, update($\alpha_u$), $N = N - 1$
14:    else
15:        return $p$
16: return $p$
```

III. From Runnables to Task Implementations

The runnable synthesis in Section **??** generates runnables while optimizing schedulability, modularity, reusability and code size. The second step of our synthesis flow maps runnables to tasks, allocates, and schedules tasks on a multicore platform. This synthesis step is today largely performed manually under the assistance of AUTOSAR design tools.

We assume that cores are scheduled according to a partitioned algorithm by local schedulers that are synchronized in time, to allow for task activations with offset (used for the preservation of execution order constraints). In the **Task view**, all the runnables (for implementing components) need to be properly mapped to tasks for execution. For simplicity, we label runnables with a single index, as in r_i (note that runnables from different components may be mapped to the same task). An ***execution order constraint*** may be defined between two runnables belonging to a component and also between runnables belonging to different components and communicating through one of the component ports. $r_i \rightarrow r_j$ denotes an execution order constraint between r_i and r_j. The closure of the execution order relation is denoted as $r_i \rightsquigarrow r_k$, meaning that there is a chain of execution order dependencies between r_i and r_k.

The task model is defined for a given core c_p as $\mathcal{T}^p = \{\tau_1^p, \tau_2^p, \ldots, \tau_m^p\}$. The core index is dropped from tasks when not required. Each task τ_j has period θ_j, activation offset ϕ_j, and is scheduled according to its priority π_j using preemptive priority-based scheduling on its core. Tasks execute the runnable functions mapped onto them according to an execution order. The worst-case response time of τ_j is denoted as w_j. A mapping relation $\mathcal{M}(\tau_j, r_s^j, k)$ defines the execution of runnable r_s^j with order k (in a sequence of runnable calls) inside the code of task τ_j (the execution index of a runnable in its task is also denoted as $k(r_i)$). Also, for simplicity, $\tau(r_i)$ denotes the task on which r_i is mapped, and $c(\tau_i)$ denotes the core on which τ_i executes. By extension, $c(r_i)$ is the core on which r_i executes. As an additional notation shortcut, the attributes of tasks and runnables are also going to be available in function form when convenient (e.g., $\phi(\tau(r_i))$ stands for the offset of the task executing r_i).

Task synthesis, i.e. **runnable to task mapping and task allocation and scheduling**, is subject to the constraints of execution order, activation rate, and schedulability and it is optimized with respect to memory requirements.

A. Constraints

The runnables (and their blocks) executed by a task must guarantee the synchronous assumption, that is, the amount of execution time requested at each activation must complete before the arrival of any following event associated with the runnable FETA. This is the ***schedulability constraint*** for the task configuration. As for the precedence constraints, if $r_i \rightarrow r_j$, the following cases apply:

- If $\tau(r_i) = \tau(r_j)$, then it must be $k(r_i) < k(r_j)$ (the execution order is enforced by the static mapping).
- If $\tau(r_i) \neq \tau(r_j)$ and $c(r_i) = c(r_j)$, then the execution order may be enforced by the priority order if $\pi(\tau(r_i)) > \pi(\tau(r_j))$. However, when $\theta_j < \theta_i$, it is possible to delay the communication between runnables by one period using a wait-free communication device at the price of additional memory.
- If $c(r_i) \neq c(r_j)$, then it must be $\phi(\tau(r_j)) > w(\tau(r_i))$ (the execution order is enforced by an activation offset).

The execution order constraints may be relaxed by adding a communication delay, which eases schedulability at the cost of memory. The commercial code generators for Simulink models allow to relax the execution order constraint by adding a Rate Transition (RT) communication buffer, which introduces a communication delay and an additional memory cost estimated as twice the size of the data communicated between the functions (for storing the values in the delay element and for the additional set of output variables). Our runnables to task mapping algorithms leverage this opportunity to improve schedulability when necessary. In addition, other communication buffers may be needed to ensure the preservation of the data flows when communicating runnables with different rates are mapped onto different tasks (even if the execution order is preserved [**?**]). Overall, the goal of the task synthesis algorithm is to find the schedulable solution with the minimum amount of memory required for RT buffers. Our two proposed algorithms try to achieve this objective in different ways.

B. Offset-based Schedulability Analysis

Tasks are activated with offset to guarantee precedence constraints among the runnables mapped onto them. Hence, the scheduling analysis consists of a set of single-core response time analysis problems with offset. The offset ϕ_i of a task τ_i is the maximum among the response times of all the predecessor runnables that have a successor in τ_i. These response times are approximated (with pessimism) with the response time of the task where they execute. Therefore, there is a circular dependency between offsets and response times (a fixed point problem) that is solved iteratively. The offsets of all tasks are initially set to 0 and response times are computed (as in [?]). Next, based on the computed response times, the task offset assignment is updated, iteratively, until response times and offsets do not change in two consecutive iterations.

The FETA $\mathcal{F}_i$ of a generic task τ_i provides the set of all the possible activation times with the corresponding execution time requests. For each core, we compute the least common multiple of all the cycle periods of the FETA of its tasks Θ_{c_i} = lcm(Θ_i). This least common multiple is the cycle period for the entire set of tasks. For each FETA $\mathcal{F}_i$ (or task τ_i), we denote the set of activation times in the scheduling period plus the largest offset $\Theta_{c_i} + \max_k(\phi_k)$ as $\{a_i(1), a_i(2) \ldots, a_i(p)\}$. When considering the task offset, ϕ_i is added to all the activation times $a'_i(k) = a_i(k) + \phi_i$. At this point, leveraging the analysis in [?], the response time for a generic request in $a'_i(k)$ of execution in the FETA of τ_i is performed by considering all the busy periods of level π_i that begin before or at $a'_i(k)$ and are not completed by $a'_i(k)$. The difference between the endpoints of these busy periods and the activation time $a'_i(k)$ is the response time for the activation in $a'_i(k)$, which has an implicit deadline at the next activation in $a'_i(k+1)$. The comparison between the response time and the deadline for each time request in the FETA allows to compute the schedulability and also the minimum slack (minimum difference between any response time and its deadline) for a task.

C. Task Synthesis Algorithms

The first task synthesis algorithm is a search heuristic extended from the solution presented for single-core platforms in [?] and the allocation heuristics in [?], [?]. The algorithm conducts a **greedy allocation with limited backtrack**, and has a post-processing step that tries to further improve the solution. The task set is constructed incrementally when a new runnables is allocated. Algorithm **??** shows an outline of the greedy heuristic.

The set of the partial execution order constraints defines a graph of runnables. Some of these runnables do not have incoming edges (are unconstrained) and are placed in an allocation list $\mathcal{L}$ sorted by their maximum local utilization $u^{\max}$ (a measure of their impact on schedulability). The algorithm iteratively picks the runnable on top of this list and tries to find a core for its execution. The core is selected based on communication affinity, that is, among the set of cores that are hosting at least one of the runnable predecessors and have a utilization lower than the threshold U_{tot}/c, where $U_{tot} = \sum_{\mathcal{C}_j} U_j$ is the sum of the utilizations of all components, and c is the number of cores. If no affine core is available, the execution core is selected among all cores.

Algorithm 3 Greedy heuristic algorithm for task synthesis

```
 1: $\mathcal{G}$ graph of all runnables with execution order dependencies
 2: $\mathcal{L} = \{\}$ is the allocation list; $\mathcal{C} = \{c_1, c_2, \ldots c_p\}$ the set of all cores.
 3: set $\mathcal{RC} = \{Rc_1, Rc_2, \ldots Rc_p\}$ as runnable-to-core allocations.
 4: set $\mathcal{TC} = \{Tc_1, Tc_2, \ldots Tc_p\}$ as task sets by core.
 5: each $Rc_i = \{\}$, $Tc_i = \{\}$ initially empty.
 6: state = firstrun;
 7: while $\mathcal{G} \neq 0$ do
 8:     $R$ = runnables in $\mathcal{G}$ with no predecessor
 9:     $\mathcal{L} \leftarrow R$
10:     $r_i$ = GetFirst($\mathcal{L}$)
11:     $\mathcal{CA}$ = ComputeAffineSet($r_i$, $\mathcal{RC}$)
12:     cur_min = -1;
13:     for all $c_k$ in $\mathcal{CA}$ do
14:         $Rc_k \leftarrow r_i$; SaveandUpdate($\mathcal{TC}$)      // try allocation
15:         min_slack = ComputeMinSlack($\mathcal{TC}$)
16:         $Rc_k = Rc_k - r_i$; Restore($\mathcal{TC}$)      // undo allocation
17:         if min_slack > cur_min then
18:             cur_min = min_slack; $c_{tgt}$ = $c_k$
19:     if cur_min > 0 then
20:         $Rc_{tgt} \leftarrow r_i$      // if schedulable
21:         $\mathcal{G} = \mathcal{G} - r_i$; Update($Tc_{tgt}$)
22:     else
23:         if state = firstrun then
24:             state = recovery      // try recovery once
25:             $\mathcal{AR}$ = ComputeAffineRunnables($r_i$)
26:             Deallocate($\mathcal{AR}$, $\mathcal{RC}$); Update($\mathcal{TC}$)
27:             $\mathcal{G} = \mathcal{G} \bigcup \mathcal{AR}$
28:         else
29:             return fail
30: return success
```

Once the list of candidate cores for allocation is selected, the runnable is tentatively allocated to each of them in turn (line **??** to **??**). When it is allocated onto a core, the tasks already mapped to the core are examined in function SaveandUpdate: If there is a task with the same period, the runnable is assigned to it. Otherwise a new task is created. Whenever a new task is added, priorities are reassigned according to the rate monotonic rule. Then, the task set is analyzed for schedulability. For each possible core allocation, the algorithm stores the minimum (or least) slack for any task in the system (computed in ComputeMinSlack). If there is at least one core that allows for a positive slack, the runnable is allocated to the core that results in the largest minimum lack (line **??** and **??**). Otherwise, the algorithm enters its recovery stage (line **??** to **??**). The recovery stage has two steps. In the first step, all the runnables that are in a dependency relation with the current runnable are deallocated and the algorithm is restarted. If one of the runnables in this list is again found unschedulable, they are deallocated again, but this time the algorithm will ignore the affinity rule when allocating them.

When a runnable is successfully allocated, it is removed from the graph and the allocation list. All its outgoing edges in the precedence graph are removed, and a new set of runnables possibly becomes available (all its predecessors are allocated) and are placed in the allocation list.

Each time a runnable is tentatively allocated and the set is analyzed for schedulability, the activation offsets are updated in the iterative procedure discussed in Section **??**. The task allocation and priority and offset assignments are constructed to enforce all execution order constraints, thereby avoiding if possible the memory cost for the rate transition buffers. However, this may prevent reaching schedulability. When the algorithm cannot find a schedulable solution, it relaxes the execution order constraints using buffer delays, and tries again to find a schedulable solution.

We also develop a **Simulated Annealing** (SA, in Algorithm **??**), as a comparison to the greedy heuristic.

In the initial configuration, each runnable is mapped to its own task. Tasks are randomly allocated to cores and scheduled using Rate Monotonic policy. At each step, A_{cur} denotes the current solution, which includes information of the runnable to task mapping, task to core allocation, and core-level task scheduling. At each iteration, a $\texttt{randomChange}(A_{cur})$ transition function computes a new candidate solution by randomly selecting one of the following transition operators:

- Randomly select a core and two tasks on that core, and swap their priorities.
- Randomly select a core and two tasks on that core, then merge the tasks and randomly set the new task priority to one of the original priorities.
- Randomly select a core and one task on that core, then randomly select a subset of its runnables and create a new task from them (with a random priority).
- Randomly select a core and one task on that core, and migrate the task to a different randomly selected core.

Solutions with cycles among runnables and tasks mapped to different cores are not allowed, because the offset update rule is not guaranteed to converge in those cases. $\texttt{existCycle}(A_{new})$ will check for the existence of cycles. After each transition, the $\texttt{memCost}(A_{cur})$ function computes the memory cost C_{new} of the new candidate solution A_{new}, and checks its schedulability using $\texttt{checkSched}(A_{new})$. If the solution is not schedulable, a penalty Δ is added to its cost. If C_{new} is better than the current cost C_{cur}, the new solution is accepted; otherwise, it is accepted with a probability that is computed by $P(C_{cur}, C_{new}, T)$ (an exponential function of the cost difference and the inverse of a *temperature* parameter T as in typical SA). For each temperature value, there can be up to K^* new solutions, and the algorithm terminates when the temperature T is lower than a minimum T^*. The best schedulable solution found A_{opt} is returned at the end.

Algorithm 4 Simulated annealing for task synthesis

```
1: Construct initial configuration.
2: while $T \leq T^*$ do
3:     while $K < K^*$ do
4:         $A_{new}$ = randomChange($A_{cur}$)
5:         while existsCycle($A_{new}$) do
6:             $A_{new}$ = randomChange($A_{new}$)
7:         $C_{new}$ = memCost($A_{new}$)
8:         is_sched = checkSched($A_{new}$)
9:         if is_sched = false then
10:            $C_{new} = C_{new} + \Delta$
11:        if $C_{new} < C_{cur}$ then
12:            $A_{cur} = A_{new}$, $C_{cur} = C_{new}$
13:            if is_sched = true then
14:                $A_{opt} = A_{cur}$
15:        else if P($C_{cur}, C_{new}, T$) > rand() then
16:            $A_{cur} = A_{new}$, $C_{cur} = C_{new}$
           $K = K + 1$
17:    $T = T * \beta$
18: return $A_{opt}$
```

IV. Experimental Results

We apply our synthesis flow to two industrial case studies and a set of synthetic examples. The experiments are run on a 3.7GHz quad-core server with 8G memory.

A. Fuel Injection Example

The first industrial case study is a portion of an automotive fuel injection control example from [?], with 58 subsystem blocks, 5 inputs and 10 outputs.

Runnable synthesis: We apply the top-down runnable synthesis method, which completes within 2 hours (the MILP formulation in the bottom-up method cannot be solved within 24 hours because of its complexity, which precludes the use of the bottom-up method for this example). By changing the desired number of runnables in Algorithm **??**, we obtain a set of runnable solutions, as shown in Fig. **??**. Each solution is measured by its modularity metric (number of runnables) and its schedulability metric (alpha ratio).

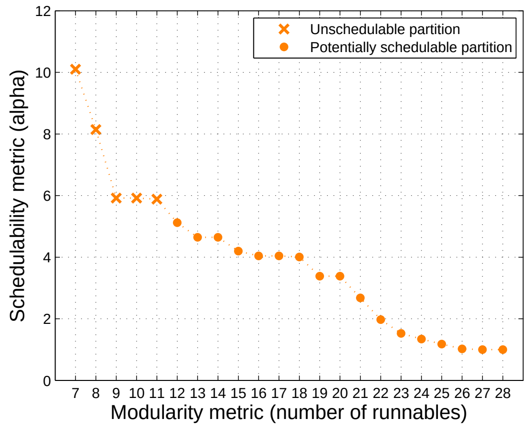

Fig. 5. Runnable synthesis with top-down method for fuel injection example

The leftmost point in Fig. **??** is the solution with optimal modularity (7 runnables, the minimum to ensure maximum reusability and minimum code size), obtained by the MILP formulation in Equation (**??**)–(**??**). This solution has an alpha ratio $\alpha_u = 10.1$, and is in fact unschedulable regardless of future task implementations (the maximum local utilization of several runnables is larger than 1). *This infeasible solution is the one that would be provided by algorithms in the literature that only optimize modularity and reusability ([**?** **?**]).*

Our top-down heuristic decomposes the runnables from the infeasible solution and improves the schedulability. For 12 runnables, $\alpha_u = 5.12$ and the system is *potentially schedulable* (i.e., the maximum local utilization of any runnable is less than 1; the actual schedulability can only be determined after the task synthesis). When the number of runnables is 28, $\alpha_u = 1$, which corresponds to the maximum schedulability.

Task synthesis: As shown in Fig. **??**, the runnable synthesis step provides a set of potentially schedulable runnable solutions with 12 to 28 runnables. We then apply the two task synthesis algorithms (greedy heuristic and simulated annealing – GH and SA in all figures) to these solutions. We randomly generate the memory cost for each possible communication buffer, and we assume a 4-core processor for task allocation and scheduling. The block execution times in this application result in an average global utilization (the total execution time requests in the hyperperiod of all runnables divided by the hyperperiod) of 60%. While this number is low, it easily results in tight local schedulability constraints for individual events, because of the large range of possible runnable periods in this example.

Runnable #	12	14	16	18	20	22	24	26	28
GH	6.3	8.7	8.3	16.8	14.5	11.1	13.8	13.4	11.5
SA	N/A	12.1	12.6	18.6	20.9	9.4	11.3	24.5	N/A

TABLE I
MEMORY COSTS OF THE TASK SYNTHESIS RESULTS FOR THE FUEL INJECTION EXAMPLE (IN KB)

Table **??** shows a summary of the results. The greedy heuristic algorithm is more effective in finding schedulable solutions (SA is not able to find a solution for the 12 and 28 runnables configurations) and produces schedulable solutions with lower memory cost, with the exception of two cases (22 and 24 runnables) where the results are comparable. The possible reason for the higher memory cost in this case is that the heuristic algorithm does not try to optimize the memory requirements for buffering communications between high priority and lower priority tasks. In terms of runtime, it is approximately 1 hour for SA and under 1 minute for GH for each case.

B. Robotics Car Example

The second industrial case study is a Simulink model of a robotics car from [**?**] that performs a path following algorithm based on a front and a lateral camera. The model has 6 inputs, 11 outputs and 28 blocks (some blocks are macro-blocks with S functions and are considered as black boxes).

Runnable synthesis: For the robotics car example, the results of both top-down method and bottom-up method are shown in Fig. **??**. The top-down method starts with a max-modularity solution (left most solution with two runnables) that is unschedulable. When the number of runnables reaches 7 in the top-down heuristic, a potentially schedulable solution is found. When the number of runnables reaches 11 in the top-down method, the maximum schedulability is reached with alpha ratio being 1. The bottom-up method starts with a max-schedulability solution (right-most solution with 8 runnables and alpha ratio being 1, obtained by the MILP formulation of the bottom-up method), and uses the heuristic to gradually merge runnables to improve the modularity. When the number of runnables decreases to 5, the alpha value increases to 2. The algorithm cannot find any 4-runnable solution that is schedulable, and therefore stops at 5 runnables.

The bottom-up method provides better results (smaller alpha ratio with the same number of runnables) because it starts

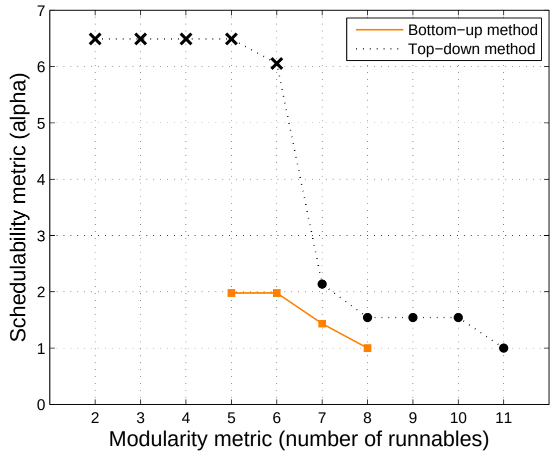

Fig. 6. Runnable synthesis (top-down and bottom-up) for the robotics case

with a solution that provides maximum schedulability and best possible modularity. Its disadvantage is longer runtimes. The bottom-up method runs 127 minutes for robotic example (most of the time is spent on the initial MILP formulation) while the top-down method takes only 7 seconds.

The task synthesis for this example is relatively easy because of the problem size. Both SA and GH are able to find schedulable solutions for all the potentially schedulable runnable generations from top-down and bottom-up methods.

C. Synthetic Examples

We also apply our algorithms to a set of synthetic examples generated by TGFF [**?**], with 10 to 50 blocks in the diagram (20 examples for each block size).

Runnable synthesis: The bottom-up method completes all examples with 20 blocks or fewer, and some of the 30-block examples. The top-down method completes within 1 hour for all examples, and in general is much faster than the bottom-up method (because of the complexity difference in the MILP formulations). We observe similar trade-offs between modularity and schedulability as in the industrial case studies, for both top-down and bottom-up methods. Fig. **??** shows the runnable synthesis results with top-down method for examples with 50 blocks. For each modularity level, the average, maximum and minimum alpha ratio values are shown. *If we apply the algorithms from literature that only consider modularity and reusability, none of their solutions is feasible.*

Task synthesis: We then apply our two task synthesis algorithms to the runnable configurations that are generated in the first step, assuming a 4-core platform. Task synthesis is performed only on potentially schedulable runnable configurations. Fig. **??** shows the comparison of schedulable percentage and relative memory cost of the two task synthesis algorithms, for different problem sizes (measured by the number of blocks in the synchronous model). The solid lines represent schedulable percentages, and the dashed lines represent relative memory costs. GH algorithm successfully schedules more than 90% of the cases, and SA schedules

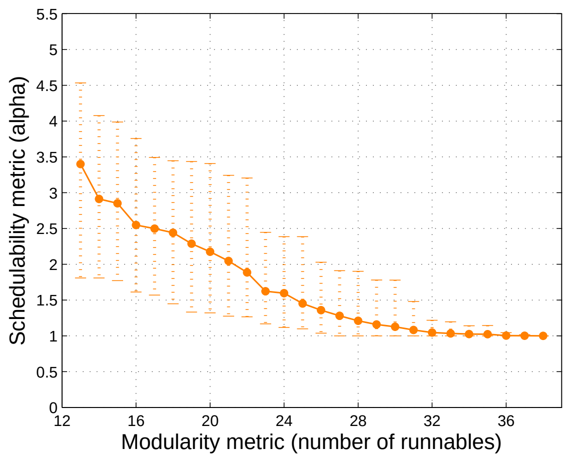

Fig. 7. Runnable synthesis with top-down method for synthetic examples

between 55% and 80% of the cases depending on the problem size. The memory cost of the solutions obtained by GH is also better than SA, in average 20% to 35% lower. In terms of runtime, GH is about two orders of magnitude faster than SA.

The high schedulability of the GH algorithm also indicates that the alpha ratio used in the runnable generation is an effective measurement of schedulability, i.e., most of the runnable configurations deemed as feasible are indeed schedulable.

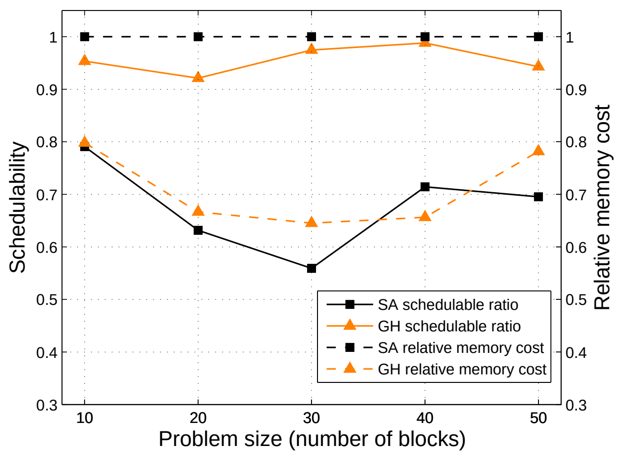

Fig. 8. Task synthesis algorithms (GH and SA) for synthetic examples

V. Conclusion

To fully unleash the power of model-based design and improve design quality, it is essential to have automated synthesis tools that can generate correct and optimal implementations from high-level functional models. In this work, we present an integrated synthesis flow for synchronous reactive models that includes two steps: a runnable synthesis step that generates runnables from SR models using a top-down method or a bottom-up method, and a task synthesis step that maps runnables to tasks and allocates and schedules tasks on multicore platforms using an effective greedy heuristic. Schedulability is being optimized throughout the flow based on the introduction of FETA and an abstract schedulability metric. Other metrics such as modularity, reusability and memory cost are also addressed, while the synchronous assumption (with its timing constraints) is preserved. Future work includes generating multiple runnable configurations for the same component and considering a more comprehensive model for code execution on multicores with the consideration of overheads.

References

[1] E. A. Lee. Cyber Physical Systems: Design Challenges. *Proc. 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008.

[2] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. *Proc. 47th IEEE/ACM Design Automation Conference (DAC)*, 2010.

[3] A. Sangiovanni-Vincentelli, W. Damm, and R. Passerone. Taming Dr. Frankenstein: Contract-Based Design for Cyber-Physical Systems. *European Journal of Control*, 18(3):217–238, 2012.

[4] AUTOSAR http://www.autosar.org.

[5] Z. Al Bayati, Y. Sun, H. Zeng, M. Di Natale, Q. Zhu, B. Meyer. Task Placement and Selection of Data Consistency Mechanisms for Real-Time Multicore Applications. Proc. of the RTAS Conference, Seattle, June 2015

[6] A. Benveniste et al. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, Jan. 2003.

[7] M. Di Natale and V. Pappalardo. Buffer optimization in multitask implementations of Simulink models. *ACM Trans. Embed. Comput. Syst.*, 7(3), April 2008.

[8] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. *11th High Assurance Systems Engineering Symposium*, 2008.

[9] C. Sofronis, S. Tripakis, and P. Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proc. 6th ACM International Conference on Embedded Software*, 2006.

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

[11] W. Peng, H. Li, M. Yao, and Z. Sun. Deployment Optimization for AUTOSAR System Configuration. In *Proc. 2nd International Conference on Computer Engineering and Technology*, 2010.

[12] M. Zhang and Z. Gu. Optimization issues in mapping AUTOSAR components to distributed multithreaded implementations. In *Proc. 22nd IEEE International Symposium on Rapid System Prototyping*, 2011.

[13] H. Zeng and M. Di Natale. Efficient Implementation of AUTOSAR Components with Minimal Memory Usage. In *Proc. 7th IEEE International Symposium on Industrial Embedded Systems*, 2012.

[14] A. Wieder and B. Brandenburg. On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks. In *Proc. 34th IEEE Real-Time Systems Symposium*, 2013.

[15] 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.

[16] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. 6th international workshop on Hardware/software codesign*, 1998.

[17] S. Edwards. An esterel compiler for large control-dominated systems. *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, 21(1):169–183, Feb. 2002.

[18] R. Lublinerman and S. Tripakis. Modularity vs. reusability: code generation from synchronous block diagrams. In *Proc. Conference on Design, Automation and Test in Europe*, 2008.

[19] R. Lublinerman and S. Tripakis. Modular Code Generation from Triggered and Timed Block Diagrams. In *Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2008.

[20] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In *Proc. 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2009.

[21] M. Pouzet and P. Raymond. Modular Static Scheduling of Synchronous Data-flow Networks: An efficient symbolic representation. In *Proc. ACM Conference on Embedded software*, 2009.

[22] The MathWorks. The MathWorks Simulink and Stateflow user's manuals. [Online] *http://www.mathworks.com*.

[23] M. Morelli et al. A Robotic Vehicle Testbench for the Application of MBD-MDE Development Technologies. *WiP session of the ETFA Conference*, 2013.

[24] A. Wieder and B. Brandenburg. "Efficient Partitioning of Sporadic Real-Time Tasks with Shared Resources and Spin Locks." In *Proc. 8th IEEE International Symposium on Industrial Embedded Systems*, 2013.

[25] P. Deng, Q. Zhu, M. Di Natale, and H. Zeng. Task Synthesis for Latency-sensitive Synchronous Block Diagram. In *Proc. 9th IEEE International Symposium on Industrial Embedded Systems*, 2014.

[26] J. Goossens, R. Devillers, and R. D. Fjgoosens. The Non-Optimality of the Monotonic Priority Assignments for Hard Real-Time Offset Free Systems. *Real-Time Systems Journal*, 13(2):107–126, Sept. 1997.