Task Synthesis for Latency-sensitive Synchronous Block Diagram

Peng Deng  
UC Riverside  
pdeng002@ucr.edu

Qi Zhu  
UC Riverside  
qzhu@ee.ucr.edu

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

Haibo Zeng  
McGill University  
haibo.zeng@mcgill.ca

Abstract—Synchronous block diagrams (SBDs) are commonly used in model-based design tools such as Simulink to capture the system behavior. In the multitask software implementation of SBDs, the execution semantics should be preserved in the value and time domains, and the task implementation should provide modular and reusable code. Previous research on component models for code generation did not consider the execution time of block implementations and the time at which outputs are produced, and did not explore the selection of task generation and scheduling based on output latencies. In this work, we propose formulations and algorithms for synthesizing SBDs into software tasks, while optimizing objectives that include timing (latency), modularity, reusability, and code size.

I. INTRODUCTION

Model-based design flows are increasingly used in the development of control applications to facilitate early verification and validation. Synchronous models are typically defined as diagrams where links connect blocks through their input and output ports. The (Mealy-type) synchronous execution semantics of a block defines a functional dependency between its outputs and inputs. When the system reaction is computed, the trigger condition of each block is evaluated. If a block is fired, the values of its output port signals are computed as a function of its input port values and its current state. Then, the block state is updated based on the inputs and the current state. These two functions are called output update and state update, respectively.

The dependency of the outputs of a block on its inputs requires that all the blocks feeding its inputs are executed in advance. In case of loops, the system behavior is defined by a fixed point equation, which in general does not guarantee a single solution. This is why loop configurations (algebraic loops) are not allowed when code is generated.

Block diagrams are often defined in a hierarchy, with components having input and output ports delegated to their internal blocks. The component is the unit of reuse and typically the atomic entity for the generation of code. For each component, two software functions are generated to compute the output and state updates for all the internal blocks according to an execution order.

The code generated for each component is executed by a software task, concurrently with tasks generated for other components (and possibly for other SBDs). The first requirement of task implementation is to preserve the synchronous assumption, which implies the preservation of flow values. Similar to hardware, a software task implementation of a synchronous model must guarantee that the functions of each component compute all reactions (outputs) before the next system event. A flow-preserving implementation guarantees that the behavior in the value domain is preserved and that the timing behavior is preserved within a bounded latency or jitter (the time to the next event).

In some control applications, the performance can be improved by selecting the implementation with the minimum latency of (a subset of) the output values. Fig. 1 shows a simple diagram with two task schedules. In the example, each block is implemented by its own task (function), and the number within the block represents the execution time of its code implementation. Two different schedules of the three tasks are shown on the right. The scheduling on the bottom right provides minimum latency for two of the three outputs. As shown later in Section III, besides task scheduling, different task generations may lead to different output latencies as well.

Fig. 1. Different task schedules result in different output latencies

To evaluate the impact of signal delays and jitter on the controls performance, the TrueTime and Jitterbug MATLAB toolboxes have been developed at the University of Lund [3]. These tools provide the co-simulation of a multitasking real-time kernel containing controller tasks and the continuous dynamics of controlled plants, to capture the impact of scheduling and task execution times on the control performance. To show how scheduling can affect the performance of control outputs even in single-rate systems, we slightly modified one of the TrueTime examples, consisting of three tasks controlling three identical servomotors, shown on the left of Fig. 2. Each task (inside the TrueTime kernel block) has a 4ms period and a variable execution rate (between 0 and 2.4ms). All controls are stable, but the quality of the control signal (and its total energy cost) is much better for the task with higher priority and lower output latency and jitter (bottom-right graph in the figure) compared with the task with lower priority (top-right graph). In this example, all control algorithms and systems are the same, to isolate the effect of the scheduling strategy. In real systems, different controls loops typically have different sensitivity to time delays. The designer must ensure that the
Automatic code generation is one of the fundamental tenets of model-based design, and methods for the correct synthesis of a (single-task) code implementation have been discussed at length [6, 1, 9]. When considering the additional need to organize the development through reusable components, issues arise on how to generate a set of functional components implementing a synchronous subsystem (hierarchical composition of a set of blocks) and how to execute them in tasks. The modularity and reusability of code implementations are quantitatively defined and investigated in [7, 8] and summarized here for convenience. Consider the example in Fig. 3. Diagram S includes three blocks, two inputs and two outputs. If S is implemented as a single functional component and reused as a black-box, the feedback connection at the top right of Fig. 3 is not allowed, leading to limitations in its reuse. In reality, there is no dependency between o1 and i2 and the feedback connection is safe since it does not result in any loop.

To improve reusability, different task implementations can be explored, such as the two tasks (functional components) c1, which includes b1 and b2 and computes o1 from i1, and c2, which includes b1 and b3 and computes o2 from i1 and i2. At the bottom right of Fig. 3, the connection from o1 to i2 becomes possible even when c1 and c2 are considered as black-boxes since no loop exists. However, b1 is part of both tasks in this new implementation. This results in a code size overhead and the need for a runtime mechanism that ensures that b1 is executed only once at each activation.

The work in [7] addresses the problem of clustering synchronous blocks for modular reuse by avoiding false input-output dependencies, with a solution for single-rate and multirate systems. The framework quantifies the modularity in terms of the number of clusters and interface functions, and explores the trade-off between modularity and reusability. In [7] the dynamic method for maximum reusability, and the step-get clustering method for best modularity and code size are proposed. In [8] the trade-off between modularity and code size is further explored. In [9], an efficient symbolic representation is proposed to simplify the modularity optimization with reusability consideration. Timing and performance are not considered in these works.

The contributions of this work are to consider timing metrics together with modularity, reusability, and code size for SBD synthesis. We propose two MILP (mixed integer linear programming) based algorithms for optimizing timing and modularity. The timing-first optimization method provides solutions with optimum timing and best possible modularity (within the timing optimum solutions), while the modularity-first method provides optimum modularity and best possible timing. Both methods guarantee optimum reusability and code size. We also propose an efficient polynomial heuristic algorithm for modularity optimization while achieving optimum reusability and code size, and an integrated MILP formulation that provides Pareto optimal solutions with respect to timing and modularity.

II. System Model

The system model we assume consists of an abstract functional model, a code implementation in functions (i.e., task generation model), and a concurrent execution of the functions through tasks (i.e., task scheduling model). Tasks are scheduled on a single-core processor. In this work, we assume the allocation of SBDs to cores is already done and focus on the task synthesis for a single core.

Functional Model. A system is a set of synchronous block diagrams (SBDs) \( S = \{S^k\} \). Each SBD \( S^k \) has a single
activation period $T^k$, and consists of a set of blocks $B^k = \{b^k_1, b^k_2, \ldots, b^k_{m_k}\}$ connected by links representing functional dependencies, a set of inputs $X^k = \{x^k_1, x^k_2, \ldots, x^k_n\}$ and a set of outputs $Y^k = \{y^k_1, y^k_2, \ldots, y^k_y\}$. For every $b^k_i$, $\text{Pred}(b^k_i)$ is the set of blocks providing inputs to $b^k_i$ (predecessors in a causal dependency graph), and $\text{Succ}(b^k_i)$ is the set of blocks using the outputs of $b^k_i$ (its successors). We use the notation $b^k_i$ to identify the block that outputs $y^k_i$. Please note that while each SBD is single-rate, the entire system may be multirate, which further justifies a multitask implementation.

**Task Generation Model.** At synthesis, a set of software tasks $\Gamma^k = \{\tau^k_i, \tau^k_2, \ldots, \tau^k_{n^k}\}$ is implemented to generate the functionality of each $S^k$. Each task $\tau^k_i$ implements a subset of blocks of $S^k$, and each block is implemented in at least one task. If a block is implemented in multiple tasks, the corresponding code is only executed once within one synchronous instant (e.g. through a modulo counter as in [7]). The generated tasks are the reusable functional components. The concepts of reusability and modularity apply to this level of the design.

**Task Scheduling and Priority Model.** We assume that all tasks generated from the same SBD are executed by a single thread according to a compile-time order. Scheduling for tasks generated from different SBDs is based on static priorities (with preemption). The compile-time order and the priority assignment may restrict reusability for a given task scheduling solution, but do not affect the reusability at the task generation level, i.e., the generated task set may still achieve maximum reusability with other schedules. As stated above, we focus on the reusability at the task generation level in this work. After the scheduling is defined, response times can be evaluated for all tasks and the blocks mapped onto them. $\tau^k_m$ denotes the response time of task $\tau^k_m$, which includes its own execution time, the time waiting for the execution of tasks from the same SBD and scheduled before it, and the time waiting for higher priority tasks from other SBDs. $l^k_{y^k_i}$ represents the delay of output $y^k_i$ (with respect to the activation time of its block). We assume (as a worst-case approximation) that all outputs are generated when the task implementing the owner block completes. Therefore, $l^k_{y^k_i}$ is equal to the response time of the task to which $b^k_i$ is mapped.

**Constraints and Objective Functions.** For each $y^k_i \in Y^k$, $l^k_{y^k_i}$ is the worst case latency within a synchronous instant, i.e. the latest possible time when the task that generates $y^k_i$ completes.

To guarantee the synchronous assumption, $l^k_{y^k_i}$ should be less than or equal to the activation period (deadline) of the SBD to which it belongs, i.e. $l^k_{y^k_i} \leq T^k$. The synthesis objective is multiple. The first result is to generate a set of tasks $\Gamma^k$ for each $S^k$ and optimize the block-to-task mapping. Then, the execution of the tasks inside the threads and the thread scheduling must be selected. The optimization objectives are multiple and include the timing, modularity, reusability, and code size of the generated task set.

**Timing.** Once the deadline constraint is satisfied, the designers may be interested in the optimization of a timing metric. We consider two possible metric functions.

The first function is to minimize the maximum latency on a (proper) subset of the system outputs $Y'$ of all SBDs (i.e. $Y' \subseteq \bigcup_k Y^k$). This metric is related to control performance. Some outputs may be control signals that are sensitive to latency, and the objective is to minimize such latencies.

$$f = \min \{ \max \{ \{l^k_{y^k_i}\} \} \}$$  \hspace{1cm} (1)

Another metric is to minimize the weighted sum of (a subset of) output latencies as in (2), where $\omega_{i,k}$ is the weight for output $y^k_i$. The motivation is a general attempt at reducing the latencies of multiple outputs.

$$f = \min \sum_{y^k_i \in Y'} \omega_{i,k} l^k_{y^k_i}$$  \hspace{1cm} (2)

Depending on the design requirements and goals, the designers may be interested in other timing metrics. For instance, they may want to maximize the minimum slack time (difference between the deadline and the latency) on a subset of outputs $Y'$, as shown below. This metric relates to better robustness or extensibility, i.e. tolerating errors in execution time estimates or accommodating new functionality. Note that if all the SBDs in the system have the same period, i.e. $T_k$ is same for any $k$, optimizing this metric is in fact equivalent to minimizing the maximum latency as in Equation (1).

$$f = \max \{ \min \{ \{T_k - l^k_{y^k_i}\}\} \}$$  \hspace{1cm} (3)

Similarly, we can have a metric to maximize the (weighted) sum of the slack time as below. Optimizing this metric can be converted to a problem of minimizing the weighted sum of output latencies as in Equation (2).

$$f = \max \{ \sum_{y^k_i \in Y'} \omega_{i,k} (T_k - l^k_{y^k_i}) \}$$  \hspace{1cm} (4)

In this work, we focus on the timing metrics as in Equation (1) and (2) as representatives.

**Modularity, reusability and code size.** The other three objectives are evaluated based on the definitions in [7, 8]. Modularity is measured by the number of tasks, and optimum modularity is achieved when the number of tasks is minimum. Optimum reusability is achieved when no false input-output dependency is introduced in the task generation. Code size is measured by adding the code size of each task, which is calculated by adding the code size of each of its blocks. Optimum (minimum) code size is achieved when no block is included in more than one task.

Next, we propose algorithms for several SBD synthesis problems with different optimization metrics.

### III. Optimization Algorithms

We use an example SBD (Fig. 4) to illustrate different task generation schemes under different optimization objectives. The number in each block denotes its execution time.

**Modularity.** Optimum modularity can be obtained by simply implementing all blocks as a single task. This solution also provides the minimum code size, but may introduce false input-output dependencies (lower reusability). In the example of Fig. 4, false dependencies would exist between $y_1$ and $x_3$, $y_2$ and $x_1$, and $y_3$ and $x_1$. This solution would also reduce the
Modularity and Reusability. In [7], the dynamic method is proposed to optimize the modularity while achieving maximum reusability. Among all the generation solutions that do not introduce false input-output dependency (hence maximum reusability), the one with the minimal number of tasks is selected. For the example in Fig. 4, the approach generates two tasks as in Fig. 5 (a): $\tau_1$ implements blocks $A, B, D$, and $\tau_2$ implements $B, C, E, F, G$. Block $B$ is included in both tasks and the code size is not optimum.

Modularity, Reusability and Code size. An approach is proposed in [8] to optimize modularity while achieving maximum reusability and minimum code size. For the example in Fig. 4, the approach generates three tasks as in Fig. 5 (b): $\tau_1$ implements $B, \tau_2$ for $A, D$, and $\tau_3$ for $C, E, F, G$. Reusability and code size are optimum, while modularity (three tasks) is worse than the previous solutions.

Timing, Reusability and Code size, then Modularity (TRCM). In this work, we evaluate the generated task set with respect to a timing metric, in addition to other objectives. We first optimize the timing objective while achieving maximum reusability and minimum code size, and then further optimize the modularity by selecting from the set of solutions generated by the first step. For the example in Fig. 4, we assume the timing objective is the sum of output latencies as in Equation (2), with the weights set to 1. The approach proposed in this paper (as discussed next) generates the solution in Fig. 6: $\tau_1$ implements $B$, $\tau_2$ for $A, D$, $\tau_3$ for $C, E, F$, and $\tau_4$ for $G$. Tasks are executed in the order $\tau_1, \tau_3, \tau_2, \tau_4$. The latencies for $y_1$, $y_2$ and $y_3$ are 9, 4 and 19. The sum of the output latencies is 32, which is the minimum value for any possible task generation and scheduling.

Modularity, Reusability and Code size, then Timing (MRCT). As a comparison to the TRCM problem, we also consider the problem of optimizing the modularity first while achieving maximum reusability and minimum code size, and then further optimize the timing. For the example in Fig. 4, this will result in the same task set as in Fig. 5 (b), and a task execution order $\tau_1, \tau_2, \tau_3$. The latencies for $y_1, y_2$ and $y_3$ are 6, 19 and 19, and their sum is 44, larger than 32 of the TRCM solution, but with improved modularity (3 tasks instead of 4).

Timing-Modularity Trade-off, Reusability and Code size. A more general problem than TRCM and MRCT is to quantitatively trade off timing and modularity, while achieving maximum reusability and minimum code size. In Section III-C, we propose an integrated MILP formulation that can provide Pareto optimal solutions with respect to timing and modularity. The solutions for TRCM and MRCT are in fact two extremes in the set of Pareto optimal solutions. In principle, the integrated formulation in Section III-C can be used for solving TRCM and MRCT, however its timing complexity makes it unsuitable for solving industrial-size problems. Therefore, we propose dedicated algorithms for TRCM and MRCT in Section III-A and III-B.

Next, we introduce an algorithm to solve the TRCM problem with the sum of latencies metric (2), and its modification for the MRCT problem. We also discuss the solutions for the TRCM and MRCT problems for the max latency metric (1).

A. Algorithm for the TRCM Problem

For every output $y_i^k$, $Z(y_i^k) \subseteq B^k$ is its input zone set, containing the block $b$ that generates $y_i^k$ and the blocks belonging to the transitive closure of its input dependencies (all the blocks on which $b$ causally depends). For every $b_i^k$, we define $O(b_i^k) = \{ y_j^k \mid b_j^k \in Z(y_i^k) \}$ (the set of all the outputs that have $b_i^k$ in their input zone) as its output set.

Our synthesis algorithm defines an execution order for all the tasks from the same SBD. $y_i^k$ is generated when all the tasks that include blocks from $Z(y_i^k)$ are completed. The execution order of the tasks also defines an output generation order (multiple outputs can be generated at the same time and arbitrarily ordered). If $y_i^k$ is before $y_j^k$ in the output generation order, $I_i((y_i^k, y_j^k))$ is defined as the interval task set, which is the set of tasks scheduled after $y_i^k$, but before $y_j^k$ is generated. The blocks in $I_i((y_i^k, y_j^k))$ are the interval block set $I_b((y_i^k, y_j^k))$. $P_{in}(y_i^k)$ denotes the set of outputs that are before $y_i^k$ in the output generation order plus $y_i^k$ itself, and $P_{out}(y_i^k)$ denotes the set of blocks scheduled before $y_i^k$ is generated.

The lemmas and theorems provided next refer to the optimality criteria for the TRCM problem, where the timing metric defined for the block outputs is the primary concern while the reusability is also maximized. These lemmas and theorems may exhibit similarities with conditions developed in other theorems and proofs from the literature regarding reusability (such as those in [8, 9], where the output timing is not considered). The theorems and lemmas below will...
facilitate the main contribution of the paper: the optimization formulations and algorithms under different objectives.

Lemma 3.1: In any optimum solution of TRCM, for any \( y^k_i \), the set of the blocks scheduled before \( y^k_i \) is the union of the input zone blocks of each output in \( P_o(y^k_i) \), i.e., \( P_b(y^k_i) = \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \).

Proof: All the input zone blocks \( Z(y^k_j) \) of a generic output \( y^k_j \) should be scheduled before \( y^k_j \) is generated. Since \( P_o(y^k_i) \) is the set of outputs that are scheduled before \( y^k_i \) in the output generation order plus \( y^k_i \) itself, the union of the input zone blocks of each output \( y^j \) in \( P_o(y^k_i) \) should be scheduled before \( y^k_i \) can be generated. That is, \( P_b(y^k_i) = \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \).

Assume there exists a block \( b \) scheduled before \( y^k_i \) but not in the above union of the input zone blocks (i.e. \( b \in P_b(y^k_i) \) and \( b \notin \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \)), we can change the block scheduling by moving \( b \) to be scheduled right after \( y^k_i \) is produced. This new scheduling will still guarantee that all the necessary blocks are already scheduled before any output is generated (outputs in \( P_c(y^k_i) \) do not need \( b \) and the other outputs have \( b \) scheduled before they are generated). Furthermore, since the execution time of any block is a positive value, this new scheduling will not increase the latency for any output, and will at least decrease the latency for \( y^k_i \). Therefore, the new scheduling solution is strictly better than the original solution, which contradicts to the optimal assumption.

Hence, we conclude that \( P_b(y^k_i) = \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \).

Theorem 3.2: If two blocks \( b^k_i \) and \( b^k_j \) have the same output set (i.e. \( O(b^k_i) = O(b^k_j) \)), then in any optimum solution of TRCM, they should be scheduled within the interval block set of two consecutive outputs \( y^k_m \) and \( y^k_n \) in the output generation order, i.e. \( b^k_j \in I_b(y^k_m, y^k_n) \) and \( b^k_j \in I_b(y^k_m, y^k_n) \).

Proof: The theorem can be proved by contradiction, using Lemma 3.1. Assume the blocks \( b^k_i \) and \( b^k_j \) are not scheduled within the interval block set of two consecutive outputs, there must exists an output \( y^k_i \) that is generated between the scheduling of the two blocks. Without loss of generality, we assume \( b^k_i \) is scheduled before \( y^k_i \) is generated, and \( b^k_j \) is scheduled after \( y^k_i \) is generated, i.e. \( b^k_i \in P_b(y^k_i) \) and \( b^k_j \notin P_b(y^k_i) \).

Based on Lemma 3.1, in any optimum solution, \( P_b(y^k_i) = \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \). Therefore, we have \( b^k_i \in \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \) and \( b^k_j \notin \bigcup_{y^j \in P_o(y^k_i)} Z(y^j) \). This means that there exists \( y^p \in P_o(y^k_i) \) such that \( b^k_i \in Z(y^p) \) and \( b^k_j \notin Z(y^p) \), which leads to \( y^p = O(b^k_i) \) and \( y^p \notin O(b^k_j) \). Then it is clear \( O(b^k_i) \neq O(b^k_j) \), which contradicts to the theorem condition. Therefore, the theorem is proven true.

Theorem 3.3: If \( Succ(b^k_i) = \{ b^k_j \} \), i.e. \( b^k_i \) is the only successor of \( b^k_j \), and there is no output directly connected to \( b^k_i \), then there exists at least one optimum solution of TRCM, in which \( b^k_i \) and \( b^k_j \) are in the same task.

Proof: Assume there is an optimum solution with \( b^k_i \) and \( b^k_j \) in different tasks \( \tau^k_m \) and \( \tau^k_n \), respectively. We prove that moving \( b^k_i \) from \( \tau^k_m \) to \( \tau^k_n \) produces another optimum solution.

It is easy to see that \( O(b^k_i) = O(b^k_j) \). Based on the Theorem 3.2, \( b^k_i \) and \( b^k_j \) are scheduled within two consecutive outputs. Therefore, the move does not change the output latencies. After the move, the set of outputs that depend on \( \tau^k_m \) will not change since the only successor of \( b^k_i \) is \( b^k_j \). Similarly, the set of inputs on which \( \tau^k_n \) depends will not be affected. Therefore, the move does not introduce false input-output dependencies through \( \tau^k_n \). The move also will not introduce false input-output relation through \( \tau^k_n \) (not needed if \( b^k_j \) was its only block), since the set of outputs that depend on \( \tau^k_n \) will not change and the set of inputs that \( \tau^k_n \) depends on is a subset of the previous. Based on these observations, the move retains maximum reusability. The move does not change the code size and schedulability, and does not decrease modularity since the number of tasks will be the same or is reduced by one.

Since moving \( b^k_i \) to the same task of \( b^k_j \) preserves the optimality of the solution, we prove that there exists at least one optimum solution, in which \( b^k_i \) and \( b^k_j \) are in the same task.

Theorem 3.4: If \( Pred(b^k_i) = \{ b^k_j \} \), i.e. \( b^k_i \) is the only predecessor of \( b^k_j \), and \( O(b^k_i) = O(b^k_j) \), and there is no input directly connected to \( b^k_j \), then there exists at least one optimum solution of TRCM, in which \( b^k_i \) and \( b^k_j \) are in the same task.

Proof: The proof is similar as in Theorem 3.3. Assume there is an optimum solution with \( b^k_i \) and \( b^k_j \) in different tasks \( \tau^k_m \) and \( \tau^k_n \), respectively. We prove that moving \( b^k_j \) from \( \tau^k_m \) to \( \tau^k_n \) produces another optimum solution.

Since \( O(b^k_i) = O(b^k_j) \), based on the Theorem 3.2, \( b^k_i \) and \( b^k_j \) are scheduled within two consecutive outputs, and therefore the move does not change the output latencies. After the move, the set of outputs that depend on \( \tau^k_m \) clearly will not change since \( b^k_i \) is a predecessor of \( b^k_j \). The set of inputs on which \( \tau^k_n \) depends on will not change either, since \( b^k_i \) is the only predecessor of \( b^k_j \) and there is no input directly connected to \( b^k_j \). Therefore, the move does not introduce false input-output dependencies through \( \tau^k_n \). The move also will not introduce false input-output dependencies through \( \tau^k_m \), since the set of outputs that depend on \( \tau^k_m \) and the set of inputs on which \( \tau^k_m \) depend on will not change (not needed if \( b^k_i \) was the only block of original \( \tau^k_m \)). Based on these observations, the move retains maximum reusability. The move does not change the code size and schedulability, and does not decrease modularity since the number of tasks will be the same or is reduced by one.

Since moving \( b^k_i \) to the same task of \( b^k_j \) preserves the optimality of the solution, we prove that there exists at least one optimum solution, in which \( b^k_i \) and \( b^k_j \) are in the same task.

Based on Theorems 3.3 and 3.4, we design our three-step algorithm as follows.

a) Step 1 (Diagram Simplification):

This step reduces the diagram size while guaranteeing the reachability of an optimum solution of TRCM. For each SBD, we calculate the input zone set \( Z(y^k_i) \) of each \( y^k_i \) using a breadth-first search (BFS). Then, we compute the output set \( O(b^k_j) \) of every block \( b^k_j \). Next, the blocks that satisfy the conditions of Theorems 3.3 and 3.4 are merged.
The inputs/outputs of the merged block are the union of the inputs/outputs of the original blocks and its execution time is the sum of the execution time of the original blocks. Merging is performed iteratively until no further reduction is possible.

b) Step 2 (Timing Optimization):

We build an MILP formulation to optimize the scheduling of the set of blocks, with respect to the timing metric. In this step, each block is considered as implemented in a single task.

For scheduling blocks within the same SBD \( S^k \), the binary variable \( u_{b_i} \) represents whether \( b_i \) is executed before \( y_j \) is generated, and \( v_{y_j} \) represents whether \( y_j \) is generated before \( u_{b_i} \). These variables determine the output generation order for an SBD and a partial execution order among blocks. For scheduling of blocks from different SBDs, the binary variable \( p_{s,h} \) denotes whether the tasks from \( S^m \) have higher priority than those from \( S^k \). \( C_{bk} \) denotes the execution time of block \( b_i \). \( C_{Sx} \) denotes the total execution time of all blocks in \( S^k \), obtained by adding the execution time of every block in it.

The MILP formulation is defined as follows. The timing objective function (formula (5)) is the weighted sum of latencies as in (2). Constraint (6) sets the values for those \( u_{y_i},v_{y_j} \) that can be statically determined from the diagram. Constraint (7) enforces the mutual consistency of the \( u \) and \( v \) variables, and (8) enforces the internal consistency of the \( v \) values (anti-reflexivity and transitivity). Constraint (9) calculates the worst-case latency of each output, which includes the execution time of the blocks that are scheduled before it (the first summation term) and the interference of higher priority tasks from other SBDs (the second summation term). The computation of the worst-case interference from higher priority tasks (10–12) uses a set of additional integer variables and the standard “big M” formulation in mixed-integer programming [12]. \( z_{m,j,k} \) is the maximum (integer) number of possible activations of the tasks generated from \( S^m \) within \( l_j \). \( m,j,k \) is the maximum number of activations that are actually generating interferences (when the tasks generated from \( S^m \) have higher priority). Finally, the constraints in (13) enforce the anti-reflexive and transitive properties of the priority assignment. Constraint (14) encodes the timing constraint of the synchronous assumption.

\[
\min \sum_{y_j \in Y} \omega_{y_{j}} l_j \quad \quad (5)
\]

\[
u_{y_i} = 1 \quad \forall b_i \in Z(y_j) \quad (6)
\]

\[
v_{y_i,y_j} \geq u_{b_i},v_{y_j} + v_{y_i,y_j} - 1 \quad \quad (7)
\]

\[
v_{y_i,y_j} \geq u_{b_i},y_j + v_{y_i,y_j} - 1 \quad \quad (8)
\]

\[
l_j = \sum C_{b_k} \times u_{b_k,y_j} + \sum C_{s,m} \times i_{m,j,k} \quad (9)
\]

\[
0 \leq i_{m,j,k} \leq z_{m,j,k} \quad (10)
\]

\[
z_{m,j,k} - M \times (1 - p_{s,m,k}) \leq i_{m,j,k} \leq M \times p_{s,m,k} \quad (11)
\]

\[
0 \leq z_{m,j,k} - \frac{l_j}{T_m} < 1 \quad (12)
\]

\[
p_{s,m,k} + p_{s,m,k} = 1; \quad p_{s,m,k} \geq p_{s,m,s} + p_{a,m,k} - 1 \quad (13)
\]

\[
l_j \leq T_k \quad (14)
\]

c) Step 3 (Modularity Optimization):

Step 2 finds the optimum scheduling of blocks with respect to the timing metric, including the partial order for scheduling blocks from the same SBD and the priorities for blocks from different SBDs. Step 3 tries to further improve the modularity of the (possibly multiple) optimal scheduling solutions found in Step 2 (these solutions can be obtained using the populate method in CPLEX).

Based on the assumption that only blocks from the same SBD can be mapped to the same task, each SBD is addressed separately in the modularity optimization. In the following, we focus on one SBD \( S \) and drop its index \( k \). We propose two solutions for this step. The first is based on an MILP formulation and suited for small-size systems, as shown in the method 3a. For large systems, a heuristic is presented in the method 3b as an alternative.

Method 3a (MILP Formulation). For each \( y_i \), Step 2 defines which blocks are scheduled before \( y_i \) is generated (i.e. \( P_{b_i}(y_i) \)) and the interval block set \( I_b(ym_i, y_i) \) for any two consecutive outputs \( y_m \) and \( y_i \) in the generation order. In our MILP formulation, we define \( G_i \equiv I_b(ym_i, y_i) \) (if \( y_i \) is the first output, \( G_i \equiv P_b(yi) \)). To keep the timing optimality during the modularity optimization, only blocks from the same \( G_i \) may be mapped to the same task.

Optimum reusability can be expressed using constraints that refer to the causal relationships among outputs and inputs. To reuse existing definitions, we associate a virtual block to each input and output, and use \( G_X \) and \( G_Y \) to denote the set of input and output (virtual) blocks, respectively. An input or output block must be mapped to the same task as the block to which the input or output belongs.

\[
b_{ri} \quad \text{denotes the i-th block in } G_r \text{, and } \tau_{r,m} \text{ denotes the generic m-th task generated for the blocks in } G_r \text{ (the task is empty if no block is mapped to it). The variable } f_{b_{ri}, \tau_{r,m}} \text{ represents whether } b_{ri} \text{ is mapped to } \tau_{r,m}. \text{ The parameter } Q_{b_{ri}, b_{sj}} \text{ indicates whether } b_{sj} \text{ causally depends on } b_{ri} \text{ (i.e. there is a path from } b_{ri} \text{ to } b_{sj}). \text{ After blocks are mapped to tasks, the causality relations among blocks will lead to causality relations among tasks.}
\]

The variable \( h_{\tau_{r,m}, \tau_{r,n}} \) represents whether \( \tau_{r,n} \) causally depends on \( \tau_{r,m} \). Finally, \( Z_{\tau_{r,m}} \) represents whether there is any block mapped onto \( \tau_{r,m} \) (the task is not empty). The formalization of the MILP problem for the minimization of the number of tasks (maximum modularity) is

\[
\min \sum_{\tau_{r,m}} Z_{\tau_{r,m}} \quad (15)
\]

\[
\sum_{\tau_{r,m}} f_{b_{ri}, \tau_{r,m}} = 1 \quad \forall r \neq X,Y \quad (16)
\]

\[
\sum_{m} \sum_{r \neq r \neq s} f_{b_{ri}, \tau_{r,m}} = 0 \quad \forall r \neq s \quad (17)
\]

\[
f_{b_{ri}, \tau_{r,m}} = 1 \quad \forall r \neq X,Y \wedge m = i \quad (18)
\]

\[
f_{b_{ri}, \tau_{r,m}} = 0 \quad \forall r \neq X,Y \wedge m \neq i \quad (19)
\]

\[
h_{\tau_{r,m}, \tau_{r,n}} \geq f_{b_{ri}, \tau_{r,m}} + f_{b_{sj}, \tau_{r,n}} + Q_{b_{ri}, b_{sj}} - 2 \quad (20)
\]

\[
h_{\tau_{r,m}, \tau_{r,n}} \geq h_{\tau_{r,m}, \tau_{r,d}} + h_{\tau_{r,d}, \tau_{r,n}} - 1 \quad (21)
\]

\[
h_{\tau_{r,m}, \tau_{r,n}} + h_{\tau_{r,n}, \tau_{r,m}} \leq 1 \quad (22)
\]
The constraints on the mapping of blocks to tasks are (17–20). Constraints (21–23) define the causality relations among tasks, and (24) requires that the block-to-task mapping does not introduce any false input-output dependencies.

Method 3b (Alternative Heuristic). The MILP formulation in method 3a computes optimal solutions at the price of a high computational complexity. For large-size problems, the efficient heuristic algorithm in Algorithm 1 provides an alternative solution for modularity optimization.

Algorithm 1: modularity_opt_heuristic(S, \{G_i\})

1: Assume each block b_i in S is initially mapped to an individual task \( \tau_i \), let \( \Gamma = \{ \tau_i \} \). Let change = true.
2: While change do
3:   While \( \exists b_i \in G_k, b_j \in G_k \) and there is no input directly connected to \( b_j \) do
4:     If schedulable_after_merge(b_i, b_j) then
5:       merge_blocks(b_i, b_j); merge_tasks(\( \tau_i, \tau_j \)).
6:   Endwhile
7:   If schedulable_after_merge(b_i, b_j) then
8:     merge_blocks(b_i, b_j); merge_tasks(\( \tau_i, \tau_j \)).
9:   Endif
10:   If \( \exists b_i \in G_k, b_j \in G_k \) and there is no output directly connected to \( b_j \) do
11:     merge_blocks(b_i, b_j); merge_tasks(\( \tau_i, \tau_j \)); change = true.
12: Return \( \Gamma \) -

The inputs are an SBD \( S \) and the partition of the blocks in \( S \) into a set of \( G_i \) from Step 2. Lines 3 to 5 apply the merge rule as defined in Theorem 3.4, but without the need for checking the output set equivalence and without restriction on the type of merging blocks (previous Step 1 only allows a merger when one of the blocks is atomic). Lines 6 to 8 apply the same merge rule as Theorem 3.3 without restriction on merging block type. Lines 9 to 11 select a block pair and checks whether the blocks can be merged. If a merger is possible, opportunities for further merging based on Theorems 3.4 and 3.3 are checked again.

The above three-step algorithm is also applicable to the cases in which minimizing maximum latency (1) is the timing optimization metric. Step 1 and 3 are the same for this metric, and only the MILP formulation in Step 2 needs to be modified according to the new objective function. Specifically, we replace Equation (5) with the formulation.

\[
\begin{align*}
\min & \quad l_{\max} \\
\text{s.t.} & \quad I_{\ell j} \leq l_{\max} \quad \forall \ell, j \in Y' 
\end{align*}
\]  

B. Algorithm for the MRCT Problem

We can leverage the above MILP formulations to address the MRCT problem. The formulation in Step 3 is modified to first optimize the modularity in task generation while achieving maximum reusability and minimum code size. We assume any block from the same SBD can be mapped into the same task (i.e., no constraints inherited from the timing optimization as in Section III-A), as long as no false input-output dependency is introduced and the system is schedulable. In [8], the same problem is solved by constructing a SAT formulation. Once the set of tasks is defined, we use the MILP formulation in Step 2 to find the optimal task scheduling sequence with respect to the timing metric.

C. Integrated Optimization

In this section, we introduce an integrated MILP formulation that explores the mapping from blocks to tasks and the scheduling of tasks to find the Pareto optimal solutions with respect to modularity and timing, while achieving maximal reusability and minimal code size. As stated before, the solutions for TRCM and MRCT are two extremes in the set of Pareto optimal solutions. However, the high computation complexity of this integrated formulation makes it prohibitive in practice. Therefore, we focus on developing TRCM and MRCT in this work and evaluating their results, while only present the integrated formulation to show its generality (without implementation and evaluation in this work).

In below, \( B^k \), \( X^k \) and \( Y^k \) denote the sets of blocks, inputs and outputs of each SBD \( S^k \). \( \tau^k_m \) denotes the \( m \)-th task generated for SBD \( k \). \( f_{b^k_i, \tau^k_m} \) indicates whether \( b^k_i \) is mapped to \( \tau^k_m \).

Maximal reusability can be expressed using constraints that refer to the causal relationships among outputs and inputs. To reuse existing definitions, we associate a virtual block to each input or output. \( b_i^k \) corresponds to an original (internal) block when \( 1 \leq i \leq |B^k| \), an input block when \( |B^k| < i \leq |B^k| + |X^k| \), and an output block when \( |B^k| + |X^k| < i \leq |B^k| + |X^k| + |Y^k| \). \( \tau^k_m \) corresponds to a task containing original blocks when \( 1 \leq m \leq |B^k| \) (the task is empty if no block is mapped to it). \( \tau^k_m \) corresponds to the input or output block with the same index (i.e. \( b^k_i \)) when \( |B^k| < m \leq |B^k| + |X^k| + |Y^k| \).

\[ Z_{\tau^k_m} \] represents whether there is any block mapped onto \( \tau^k_m \). \( G \) and \( F \) represent the modularity and the timing objective, respectively. For simplicity, we let \( C \equiv |B^k| + |X^k| + |Y^k| \). The MILP formulation is as follows.

a) Allocation constraints:

\[
\sum_{1 \leq m \leq |B^k|} f_{b_i^k, \tau^k_m} = 1 \quad \forall 1 \leq i \leq |B^k| 
\]  

\[
f_{b_i^k, \tau^k_m} = 1 \quad \forall |B^k| < i \leq |D| 
\]  

\[
f_{b_i^k, \tau^k_m} = 0 \quad \forall i \neq j, |B^k| < j \leq |D| 
\]  

Constraints (26) to (28) enforce the mapping from blocks to tasks. In particular, constraint (26) defines the mapping from original (internal) blocks to tasks, and constraint (27) defines the mapping from virtual input and output blocks to their corresponding dedicated tasks.

b) Reusability constraints:

\[
h_{\tau^k_m, \tau^k_n} \geq f_{b_i^k, \tau^k_m} + f_{b_i^k, \tau^k_n} + Q_{b_i^k, b_j^k} - 2 
\]  

\[
h_{\tau^k_m, \tau^k_n} \geq h_{\tau^k_m, \tau^k_j} + h_{\tau^k_j, \tau^k_n} - 1 
\]  

\[
h_{\tau^k_m, \tau^k_n} + h_{\tau^k_m, \tau^k_n} \leq 1 
\]  

\[
h_{\tau^k_m, \tau^k_n} = Q_{b_i^k, b_j^k} \quad \forall |B^k| < m \leq C < n \leq |D| 
\]
Constraints (29) to (31) define the causality relations among tasks, based on the causality relations among the blocks and the block-to-task mapping. Constraint (32) enforces that block-to-task mapping will not introduce any false input-output dependencies between inputs and outputs, and therefore guarantees maximal reusability. This is done by requiring that the causality relation between any input task and any output task should be the same as the causality relation between the corresponding input block and output block.

c) Timing constraints:

\[
c_{r_m} = \sum C_{b_i} \times f_{j,\tau_m} \tag{33}
\]

\[
r_{\tau_m} = c_{\tau_m} + \sum_{i,n} C_{b_i} \times x_{i,n,m,k} + \sum_i C_{a_i} \times i_{i,m,k} \tag{34}
\]

\[
x_{i,n,m,k} \leq u_{\tau_m} + \tau_{r_m} \tag{35}
\]

\[
f_{j,\tau_m} + u_{\tau_m} - 1 \leq i_{i,m,k} \tag{36}
\]

\[
0 \leq i_{i,m,k} \leq z_{i,m,k} \tag{37}
\]

\[
z_{i,m,k} - M \times (1 - p_{s',a'}) \leq i_{i,m,k} \leq M \times p_{s',a'} \tag{39}
\]

\[
0 \leq z_{i,m,k} - \frac{r_{j,m}}{T_k} < 1 \tag{40}
\]

\[
l_{y_j} = \sum m v_{j,m,k} \tag{41}
\]

\[
0 \leq v_{j,m,k} \leq r_{\tau_m} \tag{42}
\]

\[
r_{\tau_m} = M \times (1 - f_{j,\tau_m}) \leq v_{j,m,k} \leq M \times f_{j,\tau_m} \tag{43}
\]

\[
l_{y_j} \leq T_k \tag{44}
\]

\[
p_{s',a'} + p_{s',a'} = 1 \tag{45}
\]

\[
p_{s',a'} \geq p_{s',a'} + p_{s',a'} - 1 \tag{46}
\]

\[
u_{r_m} + \tau_{r_m} \geq h_{r_m} \tag{47}
\]

\[
u_{r_m} + \tau_{r_m} = 1 \tag{48}
\]

Formula (33) to (40) represent the computation of task response times. On the right hand side of (34), the first term \(c_{r_m}\) represents the execution time of task \(r_m\) (computed in (33)), the second term represents the total execution time of tasks from the same SBD and scheduled before \(r_m\), and the third term represents the time \(\tau_{r_m}\) needs to wait for higher priority tasks from other SBDs. The second term and the formula (35) to (37) are linearization of \(\sum C_{a_i} \times u_{i,m,k}\). The third term and the formula (38) to (40) are linearization of \(\sum C_{a_i} \times p_{s',a'} \times \left[\frac{r_{\tau_m}}{T_k}\right]\), using the “big M” formulation [12].

Constraints (41) to (43) relate the output latencies to the corresponding task response times. Constraint (44) enforces that output latencies are not greater than the activation period of the corresponding SBD; (45) and (46) enforce the anti-reflexive and transitive properties of the priority assignment; (47) and (48) set the scheduling among tasks from the same SBD, and ensure that it is consistent with the causality relation among tasks.

In below, formula (49) to (52) define a general multi-objective optimization problem [13] with respect to the timing metric \(\mathcal{F}\) and the modularity metric \(\mathcal{G}\). In general, there is a tradeoff between modularity and timing.

d) Objectives:

\[
Z_{\tau_m} \geq f_{h'k,\tau_m} \quad \forall i \tag{49}
\]

\[
\mathcal{G} = \sum_{m,k} Z_{\tau_m} \tag{50}
\]

\[
\mathcal{F} = \sum_{y_j \in Y'} \omega_{j,k} \times l_{y_j} \tag{51}
\]

\[
\min \left[\mathcal{F}, \mathcal{G}\right]^T \tag{52}
\]

A set of Pareto optimal solutions can be obtained by solving a set of single-objective optimization problems with constraints on the other objective. For instance, we may optimize the timing metric with modularity constraint as follows:

\[
\mathcal{G} \leq G_{MAX} \tag{53}
\]

\[
\min \mathcal{F} \tag{54}
\]

where \(G_{MAX}\) is the upper bound of the number of tasks. By varying \(G_{MAX}\), we can get a set of Pareto optimal solutions with respect to timing and modularity.

Furthermore, the timing and modularity objectives may be defined at the SBD level, i.e. with \(\mathcal{F}^k\) and \(\mathcal{G}^k\) defined as the timing and modularity of each SBD and a multi-objective optimization problem defined as the follows.

\[
\min \left[\mathcal{F}^1, \mathcal{F}^2, \ldots, \mathcal{F}^{|S|}, \mathcal{G}^1, \mathcal{G}^2, \ldots, \mathcal{G}^{|S|}\right]^T \tag{55}
\]

IV. EXPERIMENTAL RESULTS

We apply the TRCM and MRCT algorithms to two industrial case studies and a set of synthetic examples, which are run on a 2.9GHz dual-core CPU and 8G memory.

A. Test Cases

**Industrial examples.** The first example is a portion of an automotive fuel injection system [4], with 58 atomic blocks, 5 inputs and 10 outputs. Although the original model is multirate, we retained most of the graph structure, and adapted the execution time to keep the utilization similar. The second example is a Simulink model of a robotics car [11], which 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).

**Synthetic examples.** We also use TGFF [5] to generate a set of synthetic diagrams (with modifications to fit the SBD model). Each diagram has 10 to 50 blocks, with execution time and topology randomly generated.

B. Timing and Modularity Trade-off in TRCM and MRCT

**Industrial examples.** Table I shows the results of the TRCM and MRCT optimizations on the industrial examples when using two different metrics in the timing optimization step – the sum of latencies (2), where we let \(Y'\) include all outputs and the weights are 1; and the maximum latency (1), with we let \(Y'\) to be a subset of selected outputs. The MILP in method 3a is used for modularity optimization. S.L. represents the sum of latencies metric, and M.L. represents the maximum latency metric. #T represents the number of generated tasks.
Table I. Results of TRCM and MRCT for Industrial Case Studies under Two Timing Optimization Metrics

<table>
<thead>
<tr>
<th>Example</th>
<th>Sum of Latencies</th>
<th>Max Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TRCM</td>
<td>MRCT</td>
</tr>
<tr>
<td>Fuel Inj.</td>
<td>2991</td>
<td>3251</td>
</tr>
<tr>
<td>Robotics</td>
<td>805</td>
<td>1494</td>
</tr>
</tbody>
</table>

For both timing metrics, TRCM clearly provides better timing than MRCT, at the expense of worse modularity.

**Synthetic single-SBD systems.** We apply the TRCM and MRCT optimizations to a set of single-SBD systems. Each system consists of a single SBD from the synthetic examples, with 10 to 50 blocks. The MILP in method 3a is used for modularity optimization. Fig. 7 and 8 show the comparison of the relative timing and modularity between the TRCM and the MRCT results, when using the two different timing metrics. For each SBD size, we run 10 examples, with the average, maximum and minimum values shown in the figure. TRCM clearly provides shorter latencies (in average 10% to 30% reduction), at the expense of worse modularity (in average 1.5X to 2.5X number of tasks).

![Fig. 7. Timing metric comparison between TRCM and MRCT for single-SBD synthetic examples](image)

![Fig. 8. Modularity comparison between TRCM and MRCT for single-SBD synthetic examples](image)

**Synthetic multi-SBD systems.** TRCM and MRCT are also applied to a set of multi-SBD systems, where each system consists of three same-sized SBDs with different periods from the synthetic examples (the size of each SBD is again from 10 to 50 blocks). We compared the relative timing and modularity between the TRCM and the MRCT results for the two timing metrics. The results are shown in Fig. 9 and Fig. 10.

![Fig. 9. Timing metric comparison between TRCM and MRCT for multi-SBD synthetic examples](image)

Timing-wise, the single-task implementations are significantly worse – in average, 1.8 times of the MRCT and more than twice of the TRCM timing metric values. The results on multi-SBD synthetic examples show a greater gap in metric values than those obtained for single-task implementations.

The approach in [8] optimizes modularity while achieving maximum reusability and minimum code size. It provides the same modularity as MRCT (with maximum reusability and minimum code size), but it does not address timing (the focus of this paper). For comparison purposes, we implemented a simple strategy to schedule the tasks generated by the approach in [8]: we use a topological sort to determine the order among tasks generated from the same SBD and we use the Rate Monotonic policy to determine the priorities among the tasks generated from different SBDs. We compare these results with the MRCT results. For industrial examples and single-SBD synthetic examples, MRCT results are in average 20-30% better on the timing metrics. For multi-SBD synthetic examples, MRCT results are in average 30-40% better. TRCM provides an even better timing metric, at the expense of worse modularity results.

C. Comparisons with Previous Methods

We also compare the TRCM and MRCT results with single-task implementations and multitask implementations generated using the approach in [8].

For the two industrial examples and all single-SBD synthetic examples, the single-task implementations cannot provide maximum reusability except for one case, while both TRCM and MRCT always provide maximum reusability.
D. Algorithm Runtime and Heuristic vs. MILP in Modularity Optimization

In TRCM, the diagram simplification in Step 1 is a fast polynomial time algorithm. For all the examples, the runtime of Step 1 is less than 0.02 second. The timing optimization in Step 2 is fast for single-SBD systems – for industrial examples and all synthetic single-SBD examples (with 10 to 50 blocks), Step 2 takes less than 10 seconds. The runtime of Step 2 for multi-SBD systems increases quickly with respect to the size of the diagram, because of the scheduling complexity. Table II shows the runtimes for our synthetic multi-SBD examples (3 SBDs, each with 10 to 50 blocks).

![Fig. 10. Modularity comparison between TRCM and MRCT for multi-SBD synthetic examples](image)

The MILP formulation for modularity optimization in Step 3 is time consuming, while the alternative heuristic is faster, but does not provide optimality on modularity. In terms of modularity, for the fuel injection example, the heuristic generates the same number of tasks as the MILP solution when the sum of latencies is the timing metric, but a larger number of tasks when the maximum latency is the objective (22 vs. 15). For the robotics example, the heuristic generates more tasks under both timing metrics (8 vs. 4). In the synthetic examples (100 single-SBD and 100 multi-SBD cases), there are only 9 examples in which the heuristic provides worse results – most of which are multi-SBD cases with 40 or 50 blocks in each SBD. In average, the heuristic generates 19% more tasks than the MILP. The timing objective values are the same for the heuristic and the MILP.

In terms of runtimes, the heuristic completes within 0.1 second for all examples, while the MILP completes between 1 second and 91 minutes (more than 100 seconds for all examples with 40 blocks or more). The runtime of the MILP formulation depends heavily on the numbers of inputs and outputs in the SBDs. Table III shows the runtime of the MILP for synthetic single-SBD examples with 50 blocks, under different total number of inputs and outputs (roughly half are inputs and half are outputs).

<table>
<thead>
<tr>
<th># of inputs and outputs</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (seconds)</td>
<td>18</td>
<td>22</td>
<td>34</td>
<td>74</td>
<td>147</td>
<td>582</td>
</tr>
</tbody>
</table>

**TABLE III. RUNTIME OF MILP-BASED STEP 3 IN TRCM FOR SINGLE-SBD EXAMPLES WITH 50 BLOCKS**

The runtime behavior of MRCT is similar to TRCM, as it is based on the similar MILP formulations and heuristic.

V. Conclusion

We address the problem of SBD synthesis for objectives including timing, modularity, reusability and code size. We propose optimal algorithms and heuristics for two optimization problems with different tradeoffs on timing and modularity, while both achieving optimum reusability and code size. Experiments on industrial case studies and synthetic examples demonstrate the effectiveness of our approach.