Title
Model-Based Software Synthesis for Cyber-Physical Systems

Permalink
https://escholarship.org/uc/item/6df7g327

Author
Deng, Peng

Publication Date
2016-01-01

Peer reviewed|Thesis/dissertation
Model-Based Software Synthesis for Cyber-Physical Systems

A Dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Electrical Engineering

by

Peng Deng

August 2016

Dissertation Committee:

Dr. Qi Zhu, Chairperson
Dr. Sheldon Tan
Dr. Nael Abu-Ghazaleh
The Dissertation of Peng Deng is approved:

University of California, Riverside
Acknowledgments

First and formost, I’m truly grateful to my advisor Prof. Qi Zhu, who enrolled me to this lab, introduced me to this wonderful research area, patiently guided me on my research, and provided me with many valuable life and career suggestions. He is always insightful and inspiring. Many ideas in this dissertation comes from discussion with him. This dissertation wouldn’t have been possible without his guidances and supports. As his first PhD student, I am fortunate to get many close guidance from him, which helped me learned a lot not only on the knowledge domain, but also the on the skills of innovation, problem solving and critical thinking. These capabilities will continue to benefit in my future career.

I would like to thank Prof. Marco Di Natale and Prof. Haibo Zeng. Most of my PhD projects were in collaboration with them. Their guidance and insights have significantly helped me to improve my research works. I would like to thank Prof. Nael Abu-Ghazaleh and Prof. Sheldon Tan for reviewing my dissertation and for providing valuable suggestions on my research. I also would like to thank Prof. Laxmi Bhuyan, who encouraged us to develop a course project into a conference publication, and Prof. Anastasios Mourikis who provided great insights and great helps in problem formulation in one of my projects. I am also grateful to all the professors who taught me throughout the years. They helped me to established solid foundation for my research and career.

I am fortunate to have collaborated with many brilliant colleagues and researchers in my PhD career. They are but not limited to, Fabio Cremona, Hang Su, Mehmet Belviranli, Bowen Zheng. Our collaborations are always fruitful and I have learned a lot from the discussions with them. I also would like to thank my lab colleagues, Bowen Zheng, Tianshu
Wei, Hengyi Liang, Shuyue Lan and Bryan Marsh. I enjoyed working with them and they brought a lot of fun to the lab. I also deepened my knowledge and broadened my horizon through the discussions and interactions with them.

Many thanks to my best friends in UCR, Linchao Liao, Zening Li, Panruo Wu, Tian Lang, Shaocheng Wang, Vaibhav Ghadiok. We used to sharing ideas, discussing interesting technical challenges, playing basketball, traveling, having good food, watching sports games, etc. They made my life in UCR enriched. I am also very grateful to my girl friend Daisy Yu, who is a smart and wonderful person, brought lots of joy to my life. She is always supportive and understanding. She encourage me to pursue my dreams and always stand by my side in front of any difficulties.

Finally and most importantly, to my parents, who raised me up and always supported me on many life decisions. They provided me with anything I need and encouraged me to pursue my dreams, even with their sacrifices. They are my best friends and life mentors. This dissertation devotes to them.
To my parents for all the support.
ABSTRACT OF THE DISSERTATION

Model-Based Software Synthesis for Cyber-Physical Systems

by

Peng Deng

Doctor of Philosophy, Graduate Program in Electrical Engineering
University of California, Riverside, August 2016
Dr. Qi Zhu, Chairperson

Software design and implementation has become critical and increasingly challenging for cyber-physical systems (CPS) in domains including automotive, avionics, robotics, industrial automation, etc. The challenges in CPS software rise from rapid increase of system scale and complexity, close interaction between cyber components and dynamic physical environment, more sharing and contention among software functions over multicore and distributed platforms, and stringent requirements on system timing and design metrics such as performance, fault tolerance, reliability and extensibility.

During the implementation of CPS software, there are two main aspects: 1) Task generation for generating tasks from functional model. 2) Task mapping for allocating and scheduling tasks on hardware platforms. While there are a large body of works on optimizing task mapping with respect to various design metrics such as performance, extensibility and memory, task generation is carried out without much optimization of these design metrics and is detached from task mapping. Most importantly, current task generation approaches lack sufficient consideration of timing, which not only affects the optimization of design
metrics but also the functional correctness of the system. These issues often result in long software design cycles, high verification cost, and ultimately inferior and error-prone system implementations.

In this dissertation, we propose to build a model-based software synthesis framework for cyber-physical systems that automates and optimizes the generation of software tasks from functional models and the mapping of those tasks onto real-time embedded platforms in an integrated fashion, with respect to various design metrics including performance, extensibility, reusability, modularity and memory. We treat timing as a first-class citizen in this process for effectively optimizing timing-related metrics (e.g. performance, extensibility) and ensuring functional correctness. We consider a variety of practical hardware platforms, including single-core, multicore, and time-triggered or asynchronous distributed platforms. The applicability of this framework is validated by simulations on real-world CPS applications and synthetic cases.
3.3 Multi-Rate Synchronous Block Diagram Synthesis: For Automotive Systems
   3.3.1 From Multi-Rate SBD to AUTOSAR Runnables
   3.3.2 From AUTOSAR Runnables to Tasks Implementations
   3.3.3 Experimental Results

3.4 Task Period Synthesis - Codesign of Control Systems
   3.4.1 Problem Formulation
   3.4.2 Control-driven Period Optimization
   3.4.3 Comparison with Simulated Annealing
   3.4.4 Experimental Results

4 Conclusions and Future Works
   4.1 Conclusion
   4.2 Future Works

Bibliography
List of Figures

1.1 Model-based software synthesis for cyber-physical systems. 5
2.1 An Example of FSM Representation 16
2.2 An example of single rate SBD model 18
2.3 An example of multi-rate SBD model 19
3.1 An Example of FSM task implementation 28
3.2 The Implementations of General Partitioned Model 32
3.3 Extensibility for the third multi-task Implementation 33
3.4 System Action Extensibility Improvement 42
3.5 Breakdown Factor Improvement 43
3.6 Action Extensibility Improvement of Industrial Case 44
3.7 Different task schedules result in different output latencies 46
3.8 Overview of single rate SBD synthesis flow 48
3.9 Different task implementations result in different modularity, reusability and code size 52
3.10 An example of single rate SBD 52
3.11 Task implementations of Fig. 3.10 with best possible modularity and maximum reusability 53
3.12 Task implementations of Fig. 3.10 with best possible modularity, minimal code size and maximum reusability 54
3.13 Task implementation of Fig. 3.10 for TRCM 55
3.14 Fuel injection example 72
3.15 Timing metric comparison between TRCM and MRCT for single-SBD synthetic examples 74
3.16 Modularity comparison between TRCM and MRCT for single-SBD synthetic examples 75
3.17 Timing metric comparison between TRCM and MRCT for multi-SBD synthetic examples 76
3.18 Modularity comparison between TRCM and MRCT for multi-SBD synthetic examples 77
3.19 Automotive Software Supply Chain 80
3.20 Synthesis Flow for Automotive CPS ................................. 82
3.21 Runnable Generation Example with FETA notations ................ 85
3.22 Using FETA on synchronous FSM model .......................... 86
3.23 An example of multi-rate SBD ..................................... 88
3.24 Runnable Synthesis with Optimal Modularity ...................... 91
3.25 Runnable Synthesis with Optimal Reusability and Tight Timing .... 92
3.26 Runnable Synthesis with Optimal Reusability and Relaxed Timing .... 92
3.27 Runnable Synthesis Flow for Top-down and Bottom-up approaches 93
3.28 Runnable synthesis with top-down method for fuel injection example . 108
3.29 Runnable synthesis (top-down and bottom-up) for the robotics case ... 111
3.30 Runnable synthesis with top-down method for synthetic examples .... 112
3.31 Task synthesis algorithms (GH and SA) for synthetic examples ....... 113
3.32 A Simulink example of a hydraulic servomechanism (representative of a suspension control) ................................................................. 115
3.33 Performance of the electrohydraulic servo at 20ms and 100ms sampling periods 116
3.34 Functional model of an experimental vehicle ...................... 118
3.35 Quadratic Control Cost Simulation ................................. 125
3.36 Quadratic Control Cost Simulation with Delay Segmented Because of Zero-Order Hold Assumption ........................................ 126
3.37 Fitting the RMS of the control error in the Simulink electrohydraulic servo example for different sampling frequencies with an exponential decay function 127
3.38 End-to-end latency in CAN .......................................... 129
3.39 Control-driven period optimization flow ............................ 133
3.40 CDGP vs. LGP and TGP for industrial case study with exponential control cost functions .................................................. 149
3.41 CDGP vs. LGP and TGP for synthetic examples with exponential control cost function .................................................. 150
3.42 CDGP vs. LGP and TGP for synthetic examples with quadratic control cost function .................................................. 151
3.43 The number of linear partitions to be explored in theory and after pruning with different selections of $\epsilon_{\text{max}}$ .......................... 155
3.44 CDGP runtime with different selections of input size (measured by the number of blocks) .................................................. 156
# List of Tables

3.1 Results of TRCM and MRCT for industrial case studies under two timing optimization metrics ................................................................. 73
3.2 Average runtime of Step 2 timing optimization in TRCM for synthetic multi-SBD examples ........................................................................ 78
3.3 Runtime of MILP-based Step 3 in TRCM for single-SBD examples with 50 blocks ..................................................................................... 79
3.4 Memory costs of the task synthesis results for the fuel injection example (in kB) .............................................................. 109
3.5 Variable definitions for the algorithm flow .......................................................... 132
3.6 CDGP vs. RGP for industrial case study with exponential control cost function .................................................. 149
3.7 CDGP vs. SA for industrial case study (task computation time halved) . . 152
3.8 CDGP and SA with different function weights ........................................... 153
Chapter 1

Introduction

Cyber-physical systems (CPS) are the types of embedded systems whose computation and communication are tightly coupled with the physical environment. CPS has become ubiquitous around us throughout the years. The typical applications of CPS include automotive systems, avionic systems, medical robotics, industrial control systems, etc. In these CPS applications, the cyber part, which is typically a real-time embedded system, interacts with dynamically changing physical environment in a timely manner. The software in the CPS applications is the major driver for the timely interaction and is critical to system performance, robustness, reliability and security, etc. Due to the rising complexity of CPS software, the software design and implementation for CPS has become more critical and increasingly more challenging.

This section will summarize the main challenges in current CPS design, and a brief overview of our solutions to tackle these major issues, as well as discussions of related works.
1.1 Design Challenges

CPS design challenges come from rapid increase of system scale and complexity, close interaction between cyber components and dynamic physical environment, more sharing and contention among software functions over multicore and distributed platforms, and stringent requirements on system timing and design metrics such as performance, fault tolerance, reliability and extensibility. For instance, a typical modern vehicle contains about dozens even nearly a hundred of electronic control units (ECUs) [61]. From 2000 to 2010, the number of lines of code in a car has increased from 1 million to 100 millions. In a premium car, there are 2000 to 3000 singular functions that are related to software, which are combined into 250 to 300 functions used by the driver and passengers to operate the car. In Boeing’s 787 dreamliner, there are about 6.5 million lines of software code to operate its avionics and onboard support systems [15].

In recent years, model-based design paradigm is becoming widely used for CPS design due to its capability to support early design verification/validation through formal functional models [38][57][62]. During the software implementation of CPS, there are two main aspects: 1) task generation for generating software tasks from functional models, and for synthesizing tasks specifications (e.g. task periods for control-centric applications). 2) task mapping for allocating and scheduling tasks on hardware platforms. Despite the advancement of existing synthesis tools, there are still two major issues in current approaches:

- Gap between task generation and task mapping: while there is a large body of work on optimizing the mapping of a given software task set onto a hardware platform with respect to various design metrics such as performance, extensibility and reusability,
the generation of software tasks from functional models is carried out without much consideration of those metrics and detached from task mapping;

- Lack of timing consideration in task generation: current task generation approaches lack sufficient consideration of timing, which not only affects the optimization of design metrics but also the functional correctness of the system.

The above challenges are intensified in automotive system design, whereas an intermediate runnable level is introduced to facilitate automotive supply chain and improve software reliability and scalability. Besides gap between task generation and task mapping, there is another gap between runnable synthesis and task synthesis. Furthermore timing consideration is certainly being overlooked in runnable synthesis stage.

These issues above often result in long software design cycles, high verification cost, and ultimately inferior and error-prone system implementations.

1.2 Overview

In this dissertation, we propose to build a model-based software synthesis framework for cyber-physical systems that automates and optimizes the generation of software tasks from functional models, and the mapping of those tasks onto real-time embedded platforms in an integrated fashion, as well as the optimization of task period for control applications. We consider various design metrics including performance, extensibility, reusability, modularity, code size and memory. We treat timing as a first-class citizen in this process, which is critical for both optimizing timing-related metrics (i.e. schedulability, extensibility, robustness) and ensuring functional correctness. We consider a variety of practical hardware
platforms, including single-core, multicore, and time-triggered or asynchronous distributed platforms.

Among various functional modeling languages, we focus on the synthesis for synchronous reactive (SR) models, which dominate the current practice for modeling control-centric CPS applications in domains such as automotive and avionics (the popular Simulink and Stateflow toolset [2] and the SCADE suite [3] are based on SR semantics). An SR functional model is typically represented by a network of functional blocks in a synchronous block diagram (SBD), where some blocks may be macro blocks that include lower-level blocks and captured by lower-level SBDs. Functional blocks can be of two types: dataflow blocks or extended finite state machine (FSM) blocks. For macro blocks and FSM blocks, existing synthesis approaches conduct either single-task synthesis or limited multi-task generation without much optimization, and mostly for single-core platforms. Our proposed framework explores multi-task generation of macro blocks (SBDs) and FSM blocks for optimizing various objectives on a variety of hardware platforms. After task generation, our proposed framework further explores assignment of task periods for optimizing control performance.

Fig.1.1 provides an overview of the work we have carried out so far for model-based software synthesis of CPS. Specifically, we have addresses the following problems:

- We address the problem of synthesizing FSMs in SR models, and propose algorithms for multi-task generation of synchronous FSMs to optimize system robustness and extensibility [76].

- We address the problem of synthesizing single rate SBDs in SR models on single-core platforms, and propose algorithms for multi-task generation of SBDs to address
system latency, modularity, reusability, and code size [21].

- We address the problem of synthesizing multi-rate SR models on multi-core platforms in automotive domain, and propose formulations and algorithms for addressing system schedulability, modularity, reusability, and memory usage [19].

- We address the problem of synthesizing task period for electronic control applications on distributed platforms in automotive domain, and propose formulations and
algorithms to optimize control performance while guaranteeing system schedulability constraints [20].

1.3 Related Works

As stated in [20], [19], [21], [76], a number of formal definitions belong to the general category of SR models, such as the Signal, Lustre, and Esterel [10]. 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 [26, 55].

The Simulink/Stateflow toolset [2] 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 code generators allow a single task and a fixed-priority multitask code generation options. Single-task implementations preserve the model semantics at the cost of a possible impact on task schedulability.

The problems of generating multi-tasks implementation for FSM models have been studied in [22]. It is observed that a single-task implementation as those in the synthesis tools may lead to unnecessary activations, and therefore reduce the system schedulability and cause memory overhead on communication buffers. By generating multiple software
tasks according to a *partitioning* of the FSM based on the periods of the trigger events, the system schedulability and memory usage may be improved. Different task implementation models are discussed. However, no approach is provided for quantitatively comparing them. Schedulability analysis of Stateflow models or, in general, of real-time task models derived by synthesis from a state machine formalism is discussed in [71] based on an analogy with the digraph task model [67], previously proposed for the analysis of concurrent real-time tasks with branching.

In the task generation of synchronous block diagrams, when considering 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 [43, 44]. The work in [43] 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 [43] the dynamic method for maximum reusability, and the step-get clustering method for best modularity and code size are proposed. In [44] the trade-off between modularity and code size is further explored. In [55], an efficient symbolic representation is proposed to simplify the modularity optimization with reusability consideration. Timing and performance are not considered in these works. 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 [52], where a branch-and-bound algorithm is presented. In [51], an MILP (Mixed Integer Linear Programming) formulation is provided. In [45], the Firing Time Automaton formalism is proposed for expressing firing times of multirate components. The Prelude synchronous language [27] includes rules and operators for the selection of a mapping onto platforms with Earliest Deadline First (EDF) scheduling, including symmetric multicore architectures. The enforcement of the partial order of execution that is required by the SR model semantics is obtained in Prelude by a deadline modification algorithm. Communication among nodes executing at different rates is realized using the protocol and data structures defined in [66] (an extension of [14] which is optimal with respect to the number of required buffers for a system containing only periodic nodes).

In automotive domain, the software synthesis flow could be more complex. In particular, in AUTOSAR 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. The problem of mapping AUTOSAR runnables onto ECUs in a distributed system with the objective of bus load minimization is discussed in [54, 74], with solutions based on a genetic algorithm. The anal-
ysis of real-time communication mechanisms based on spin locks for AUTOSAR runnables in multicores is presented in [69], and mechanisms for a flow-preserving AUTOSAR implementation of synchronous signals are presented and compared in [72].

Another important aspect in task generation is to synthesize the configurations for given task implementation. Specifically, in many control-centric CPS applications, the performance of the control function is significantly affected by its sampling period and end-to-end control delay. Finding the optimal configuration with respect to schedulability constraints is a very complex optimization problems. As stated in [20], several approaches have been proposed in the literature to optimize control performance under schedulability constraints. In [64], tasks periods are selected to optimize control performance on a uniprocessor system with the EDF unit utilization bound. The work is extended in [11] by considering the impacts of control delays on performance. In [73], task scheduling is studied for feedback control systems implemented on uniprocessor platforms. Task periods are assigned to balance robustness, schedulability and power consumption. In [25], task periods are assigned to optimize control performance on a uniprocessor platform and utilization bound is used for schedulability analysis. The codesign of controller-server is studied in [5], where periods are selected for servers to minimize bandwidth while guarantee stability. In [37], the impact of delay on control performance is investigated and methods are proposed to derive a delay-frequency interface that enables the verification of control performance properties. The study of the optimal priority-based network scheduling with respect to controls performance can be found in [13]. In [58], a genetic algorithm is used to select task periods and scheduling to optimize the control performance on a distributed plat-
form. An extension of the algorithm applies to FlexRay networks [59]. In [60], task periods are dynamically adapted to optimize the performance of controls within the schedulability boundary for a uniprocessor platform. In [29], task sampling periods and bus scheduling are synthesized for FlexRay-based platforms. The sampling periods are enumerated from a set of possible values and the problem is solved by integer linear programming (ILP). In [6], both control quality and robustness are considered in the exploration of sampling periods, system scheduling and control synthesis. The controller periods are optimized by using the coordinated search method combined with the direct search method. Most of the previous approaches either focus on uniprocessor systems, or use randomized search methods (e.g. genetic algorithms, simulated annealing) to explore period assignments. The randomized search methods are prone to end up in local minima, and need constraints to be encoded as part of the objective function (often in an ad-hoc fashion) or as a post-filtering of the solution generated at each step. Furthermore, randomized search methods not only do not provide guarantee of optimality, but do not give any information on the quality of the solution they compute and the designer cannot know how far it is from the optimum. Thanks to the duality theorem, the solvers for convex optimization formulations such as geometric programming (GP) can provide at any step an estimate of the optimality gap, which is the worst case distance between the current solution and an optimal solution (if it exists). In our case, this allows to measure the quality of the solutions of the GP optimization step (an inner loop in our algorithm). Even though this does not provide any guarantee for the optimality gap of the final solution, it provides a feedback on the quality of the intermediate results of the algorithm. In [29], mathematical programming based method is used to
explore optimal message and task scheduling on distributed systems. However, the solution presented in [29] is tailored to a specific set of time-triggered systems (with a FlexRay bus) in which tasks are scheduled without preemption and messages are transmitted in a strict cyclic fashion. As a consequence, the objective of [29] is not really period selection (which is enumerated since only multiples of the communication cycle are allowed) but rather time-triggered scheduling using MILP. Many control systems are however still deployed on priority-based scheduled networks and CPUs, such as the CAN (Controller Area Networks) bus and ECUs with AUTOSAR based OS in automotive systems. For these systems, period enumeration is not possible. Furthermore, the response time formulations have periods on the denominator, making an MILP formulation of the period optimization impossible.

Metrics are also proposed to measure task implementations. The breakdown utilization was introduced in [40] to characterize the ability of the rate monotonic scheduling algorithm to meet the task deadlines. The breakdown utilization is defined as the task set utilization when the execution times of all tasks in a task set are scaled (by the same factor) to the point at which a deadline is first missed. Intuitively, a larger breakdown factor allows a wider selection of implementation platforms (e.g. different processor speeds for cost/performance trade-off) and makes the system more robust with respect to timing variations.

Another widely studies metric is extensibility. There has been a number of works addressing extensibility at the level of software tasks [70, 49, 56, 9, 31, 33, 32, 75, 77]. In [70, 49], extensibility is studied at the task level with no explicit system synthesis. In [56], sensitivity analysis is conducted for priority-based distributed systems and the syn-
thesis problem is partially addressed by using genetic algorithms for optimizing priority and period assignments. In [9], task allocation and priority assignment are explored to optimize a metric of extensibility in which all task times are scaled by a constant factor. In [31, 33, 32], a general definition of extensibility at task level is presented and a genetic algorithm is proposed. In [75, 77], an algorithm combining heuristics with mixed-integer linear programming is proposed to optimize a task-level extensibility metric that measures how much the execution times of software tasks can be increased without violating schedulability constraints.

1.4 Contributions

The main contribution of this work includes:

• Developed design methodologies that integrate task generation and task mapping in a holistic flow. The proposed approach automates and optimize the task generation from functional model and task mapping into hardware platforms.

• We modeled various timing based metrics and optimize them throughout the synthesis flow. Other software engineering based metrics such as modularity and reusability are considered together with timing-related metrics, for instance, modularity, reusability, code size, etc.

• Developed a co-design frameworks that optimizes CPS control performance while guaranteeing the system timing constraints.
In the rest of the dissertation, the system models at different implementation levels of CPS applications are introduced in Chapter 2. The synthesis flow for various functional models are present in Chapter 3, which include synthesis of FSM model and dataflow model, as well as synthesis of task period for control applications. Chapter 4 will conclude our work and discuss future directions.
Chapter 2

Background

In this chapter, we present the system models for cyber-physical systems at different abstraction levels. We first introduce the functional models that capture the functionality of the systems at highest abstraction level. Then, we will introduce AUTOSAR runnables, which is an automotive specific layer generated from functional model and represents basic software modules. Next, we will discuss how tasks are generated from either runnables (i.e. for automotive systems following AUTOSAR) or directly from functional models (e.g. other CPS). Finally, we will discuss how software tasks are mapped to hardware platforms and are statically scheduled on cores.

2.1 Functional Models

In this paper, we primarily focus on synchronous reactive model as it is widely used and dominate the current development of control-centric CPS applications. In synchronous reactive (SR) model, the system takes continuous inputs from physical environment and
maintains an ongoing outputs with respect to the environment changes, therefore it makes SR model a natural fit for cyber-physical applications, where the cyber components “react” in real-time with respect to the changes in physical environment.

An SR functional model is typically represented by a network of functional blocks in a synchronous block diagram (SBD), where some blocks may be macro blocks that include lower-level blocks and captured by lower-level SBDs. Functional blocks can be of two types: extended finite state machine (FSM) blocks or dataflow blocks (macro blocks). In this dissertation, we consider 1) Synchronous finite-state machine model, which is used to implement FSM blocks. 2) Synchronous block diagram, which is used to implement macro blocks.

Functional blocks in SR model are in general multi-rated, which is consistent with the practical applications. For instance, in automotive systems, off-the-shelf sensors operates in different rates due to technology constraints or system requirements, and the usage of multiple control functions is governed by different control laws. Merging control flows with different rates and communication with oversampling/undersampling is commonplace.

2.1.1 Synchronous Finite-State Machine

We use the similar notations as in [22] to represent a synchronous (Mealy type, inspired by Stateflow) finite state machine. An FSM is defined by a tuple $(S, S_0, I, O, E, T)$, where $S = \{S_0, S_1, S_2, \ldots, S_l\}$ is a set of states, $S_0 \in S$ is the initial state, $I = \{i_1, i_2, \ldots, i_p\}$ and $O = \{o_1, o_2, \ldots, o_q\}$ are the input and output signals. $E$ is a set of trigger (or activation) events. Each event $e_j$ is generated by the value change of a signal, and may only occur with a period $t_{e_j}$ that is an integer multiple of a system base period $t_{base}$ (i.e. $t_{e_j} = k_{e_j} \cdot t_{base}$).
At each time instant $k \cdot t_{e_j}$, the event may or may not present (in which case the system may stutter). Finally, $T$ is a set of transition rules. Each transition $\theta_j \in T$ is a tuple $\theta_j = \{S_{s_j}, S_{d_j}, e_{\theta_j}, g_j, a_j, \gamma_j\}$, where $S_{s_j}$ is the source state, $S_{d_j}$ is the destination state, $e_{\theta_j} \in E$ is the trigger event (different transitions may be triggered by the same event), $g_j$ is the guard condition, $a_j$ is the action, and $\gamma_j$ is the evaluation order among the transitions that start from the same state. When two or more transitions from the same source state are enabled at the same time, the transition with the lowest order is taken.

Fig. 2.1 shows an example FSM. The periods of $e_1$ and $e_2$ are 2ms and 3ms respectively. For each transition, the associated trigger event and action are shown, along with the transition priority (in red and bold) and the execution time of the action (in blue and italic). The guard condition is omitted.

In this dissertation, we do not consider the case in which transitions are activated by a logical expression evaluated on multiple events, or events generated by transitions. The Stateflow semantics also allow more advanced constructs such as concurrent states, super-states, entry/exit actions, while actions and join transitions. In several cases, a hierarchical
FSM can be flattened and our methods can apply. The conditions for the translation into flat FSMs are outlined in [63] and the composition rules can be found in [39].

2.1.2 Single-Rate Synchronous Block Diagram

In this model, a system is a set of synchronous block diagrams (SBDs) \( S = \{S^k\} \) [21]. 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 \( X^k = \{x^k_1, x^k_2, \ldots, x^k_{p_k}\} \) and a set of outputs \( Y^k = \{y^k_1, y^k_2, \ldots, y^k_{q_k}\} \). 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 \( \hat{b}^k_j \) to identify the block that outputs \( y^k_j \). \( C_{b^k_i} \) denotes the execution time for block \( b^k_i \). \( C_{s_k} \) denotes the total execution time of all blocks in \( S^k \), obtained by adding the execution time of every block in \( S^k \). Note that different SBDs might have different periods, but all blocks in the same SBD have the same period. \( T_i \) denotes the period for SBD \( S^i \).

Fig. 2.2 shows an example of single rate SBD. The dashed large block denotes the block diagram which groups a set of functional blocks. The number on upper right corner in each block denotes its execution time. The period of SBD is 50 as denoted on the upper left corner of dashed block. In general, the SBD period should be larger than \( \sum C_{b^k_i} \) to guarantee the functional correctness of the system.
2.1.3 Multi-Rate Synchronous Block Diagram

In this model, a system is a synchronous block diagram (SBD) composed of a set of subsystems or components $S = \{C_j\}$ [19]. Each component $C_j$ consists of a network of blocks $B_j = \{b^j_1, b^j_2, \ldots, b^j_{m_j}\}$ connected by links representing functional dependencies, a set of inputs $X_j = \{x^j_1, x^j_2, \ldots, x^j_{p_j}\}$ and a set of outputs $Y_j = \{y^j_1, y^j_2, \ldots, y^j_{q_j}\}$. The output $y^j_k$ depends on the input $x^j_p$, denoted as $D(y^j_k, x^j_p)$, if there is a path of links and blocks (indicating a causal dependency) between $x^j_p$ and $y^j_k$. $X(b^j_k)$ and $Y(b^j_k)$ denote the set of input and output ports directly connected to block $b^j_k$, respectively. Each block $b^j_i$ has an activation period $t^j_i$ and a WCET $\gamma^j_i$. Whenever there is an input-output dependency between a sender block $b^j_i$ and a receiver block $b^j_k$ for which the output update function depends on the input values ($b^j_k$ is of type feedthrough), there is an execution order constraint between the two blocks (and their implementations) $b^j_i \rightarrow b^j_k$. For every $b^j_i$, $\text{Pred}(b^j_i)$ is the set of its predecessors, and $\text{Succ}(b^j_i)$ 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. 2.3 with 6 blocks, labeled A to F, is used as a running example. The worst-case execution time and periods of the blocks are shown in the bottom of each block in the parenthesis. Functional dependencies between blocks are shown as links.

![Figure 2.3: An example of multi-rate SBD model](image)

2.2 Runnables

Since 2003, in order to promote software reusability and scalability among different vehicle platforms, majority of automotive companies established AUTOSAR (AUTomotive Open System ARchitecture) [1], which serves as an open and standardized software architecture for automotive systems.
In the AUTOSAR software supply chain, a code implementation is generated from functional models into 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. The AUTOSAR runnable will then be delivered to integrators, i.e. automotive makers and the final automotive software are integrated and implemented based on runnables.

In this dissertation, we mainly consider runnables implemented from multi-rate SBD model (i.e. Section 2.1.3), which is the general case for automotive systems. An implementation from runnables view $R_j$ consisting of a set of runnables $R_j = \{r_{j1}, r_{j2}, \ldots r_{jp}\}$ is provided for each component $C_j$. Each runnable is the code implementation of a subset of the blocks in $C_j$, denoted as $BI(r_{jk}) = \{b_{jk1}, b_{jk2}, \ldots b_{kjs}\}$ with the requirement that the union of all the runnable blocksets covers all the blocks in $C_j$. 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 $r_{jk}^i$ has a base period $\phi_{jk}^i$ corresponding to the greatest common divisor of all the blocks in $BI(r_{jk}^i)$. A set of partial execution order constraints $r_{jk}^i \rightarrow r_{jk}^j$ 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, $X(r_{jk}^i)$ and $Y(r_{jk}^i)$ denote the set of input and output ports accessed by runnable $r_{jk}^i$ with $X(r_{jk}^i) = \bigcup_t X(b_{kt}^i)$. 
Note that at the runnable level, there is no implication of underlying hardware platforms or task scheduling or priorities, therefore its timing behavior is difficult to measure at this level. In this dissertation, we propose formulations to measure runnable timing and the details will be discussed in later sections.

2.3 Tasks

In the design of CPS, tasks are generated either from runnables (i.e. automotive) or directly from functional models. A software task is essentially a thread that runs on embedded platforms. In CPS applications, depending on the original functional model, a single task $\tau_i$ can implement a set of transitions (FSM model) $\tau_i = \{\theta^i_1, \theta^i_2 \ldots \theta^i_k\}$ or a set of functional blocks (SBD model) $\tau_i = \{b^i_1, b^i_2 \ldots b^i_k\}$ or a set of runnables (SBD model) $\tau_i = \{b^i_1, b^i_2 \ldots b^i_k\}$. Because of the reactive nature of original functional model, a task implemented from a set of transitions/blocks/runnables will be “reactive” on a period of $T_{\tau_i}$. To make sure each transitions/block/runnable will not miss any activation, the task period must to be set as the greatest common divisor (GCD) of the periods of the triggering events in transitions, or activation periods in blocks/runnables.

A task implementation in CPS can be either single task or multiple tasks (threads) sharing the same memory space. While single task benefit from its simplicity and modularity, it suffers on system timing and therefore can further degrade system performance, reusability, security and reliability, etc. In this dissertation, we mainly focus on and promote the multi-task implementations, the detailed reasoning and explanation will be discussed in later sections.
A multi-task implementation consisting of a set of tasks $\tau = \{\tau_1, \tau_2, \ldots, \tau_k\}$ synthesized from functional models/runnables. Tasks may have causality relations among them that is inherited from original functional model. The causality relation denotes the data dependencies among tasks. Multiple decisions have to be made upon the task synthesis stage: task generation, task allocation and task scheduling. Task generation is the process of grouping functional blocks/transitions into tasks. Task allocation is the process of assigning tasks to processors/cores, and task scheduling is the process to determine the execution order/priorities when they are assigned to the same core. In CPS applications, these decisions are often made at design time. Tasks are statically linked to cores and are assigned with fixed priority $p_\tau$. There are other dynamic scheduling methods, e.g. Earliest Deadline First [35], but they may not be the best candidate for systems with strong timing constraints that need more deterministic behavior. They are out of the scope of this dissertation.

In a multi-task implementation, tasks can be scheduled using either preemptive scheduler or non-preemptive scheduler. By using preemptive scheduler, lower priority tasks that are in the middle of execution can be preempted (interrupted) by higher priority tasks assigned in the same core, whereas by using non-preemptive scheduler, a task that is currently running can always be allowed to finish regardless of invocation of higher priority tasks. For either scheduler, tasks may take longer than its worst case execution time to finish because of the interference from higher priority tasks. The time elapses between task release time to task finish time is called response time $r_\tau$.

A deadline $d_\tau$ of a task is the time by which a task must complete its execution.
In general, there are *soft deadlines* and *hard deadlines*. For soft deadline, missing a deadline is considered a degradation of performance and it is better for a task to finish before the deadline. For hard deadline, missing a deadline is considered as a system fail and may be catastrophic to the system. CPS applications in general use a combination of hard deadline components (e.g. powertrain systems) and soft deadline components (e.g. infotainment system in cars). In this dissertation, we mainly considered timing critical component in which tasks have **hard deadlines**. Throughout the proposed design flow, we explicitly address system timing, because in hard real-time systems, timing behavior affects not only the performance but also the functional correctness of the system.

Because of the multi-rate nature of SR model, the multi-task implementation from SR model is multi-rated. In this case, tasks with direct dependencies may have different rates. To guarantee the correct task execution, rate transition blocks should be added. For instance, the commercial code generators for Simulink models allow to relax the execution order and rate 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).

### 2.4 Hardware Platform

In this dissertation, we mainly focus on three types of hardware embedded platforms: single core, multi-core and distributed platforms.
**Single Core:** There is only one core in the system, or the software component/functional model is only implemented and allocated on one core of a multi-core or distributed platforms. There can be multiple tasks running on this core, and tasks communicate through shared memory space.

**Multi-Core:** There are multiple cores in the system, and cores are assumed to be on a single processor. Tasks among different cores communicate through shared memory space. In this dissertation, we assume the timing overhead of accessing shared memory is negligible compared to other timing elements. This implies that data produced by one tasks is immediately available for other depending tasks.

**Distributed Systems:** There are multiple processors in the system, and each processor could have one core or multiple cores. The hardware model within one processor is the same as those in single core and multi-core model. Processors are connected through shared bus, and tasks from different processor communicate by sending a message through the bus. Specifically, in this dissertation, we consider the automotive systems in which a set of electronic control unit (ECUs) are connected via controller area network (CAN) bus. The details of task models and message passing models on this hardware platform are discussed in later sections.
Chapter 3

CPS Software Synthesis Flow

In this chapter, we present our software synthesis flow for cyber-physical systems. We discuss our motivations and present design metrics as well as synthesis algorithms and methodologies. The synthesis flow mainly focuses on two parts: 1) synthesize software tasks from functional models, they can be either synchronous finite-state machine or synchronous block diagram, and 2) synthesize task periods for embedded control applications, whereas control performance is optimized while guaranteeing system timing constraints.

3.1 Synchronous Finite-State Machine Synthesis

In the implementation of Synchronous FSM models, current tools typically generate a single task implementation. For example, the languages supported by SCADE (Esterel, Lustre, and SyncCharts [10]) are typically implemented as a single executable that runs according to an event server model. In Simulink, Embedded Coder/Simulink Coder [2], generate a single periodic task for each Stateflow block (FSM). Every time this task is activated,
it checks for active trigger events and processes them. To make sure it will not miss any trigger event, the task is executed at the greatest common divisor (GCD) of the periods of its trigger events. As proposed in [22], single-task implementation may lead to unnecessary activations and therefore reduce the system schedulability and cause memory overhead on communication buffers. Multi-task implementation based on the period of trigger events may improve system schedulability and memory usage.

In this section, we propose methods to quantitatively analyze and compare task implementations with respect to a breakdown factor and an action extensibility metric. We then present algorithms to generate multi-task implementations which optimize the proposed metrics.

The breakdown factor is defined based on the concept of breakdown utilization (introduced in [40]) as the maximum scaling factor of task execution times that allows to retain feasibility (in our work, the factor applies to actions in FSMs). Intuitively, a larger breakdown factor allows a wider selection of implementation platforms (e.g. different processor speeds for cost/performance trade-off) and makes the system more robust with respect to timing variations.

Action extensibility follows the definition of task extensibility (proposed in [75]) as the maximum amount by which the execution time of each single action in the FSMs may increase without violating the system timing constraints. Optimizing action extensibility allows adding future functionality or upgrading existing ones without a major redesign cycle. This is imperative for large-volume and long-lifetime systems.
3.1.1 FSM task implementation Model

As stated in [76], we consider implementing a set of FSMs $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$ on a single embedded processor with a set of tasks $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_k$ has period $\psi_k$ and priority $\pi_k$. If a transition $\theta_j$ (and the corresponding action $a_j$) is implemented in $\tau_k$, it is denoted as $\mu(\theta_j) = \tau_k$. We assume that only transitions from the same FSM may be implemented in the same task.

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

To illustrate the different task implementations and their impact on quantitative metrics, we use the FSM in Fig. 3.1 as an example to compute the breakdown factor and the action extensibility.

a) Single-task implementation

In a single-task implementation, all transitions from the same FSM $F_k$ are implemented in a single task $\tau_k$, i.e. $\forall \theta_j \in F_k, \mu(\theta_j) = \tau_k$. This is the approach adopted by the
the Simulink Coder generator, and is called *baseline implementation* in [22]. The period of \( \tau_k \) is the GCD of the periods of all trigger events, i.e. \( \psi_k = GCD(t_{\theta_j}) \) where \( \theta_j \in F_k \). To ensure that every transition completes before the next set of inputs is processed (by a new instance of the same task), the largest action execution time should be no larger than the period, i.e. \( \max(C_{a_j}) \leq \psi_k \) where \( C_{a_j} \) is the execution time of \( a_j \).

For a FSM in Fig. 3.1, a single task \( \tau_k \) implements all transitions. The task period \( \psi_k \) is the GCD of event periods 2\(ms \) and 3\(ms \), i.e. 1\(ms \). All transitions must complete within the task period, therefore the breakdown factor is \( \psi_k / \max(C_{a_j}) = 1/0.5 = 2 \). The extensibility of action \( a_1 \) is \( \psi_k / C_{a_1} = 1/0.4 = 2.5 \). Similarly, the extensibility of \( a_2 \) to \( a_5 \) are 5, 3.33, 2 and 2.5, respectively.

b) Multi-task implementation

In [22], three task models are presented for multi-task implementations of synchronous FSMs: the partitioned model, the mixed-partitioned model and the deferred output update model. The deferred output update model is based on separating the state
update function and the output update function into different tasks. This may require significant development and verification efforts and cause extra overhead if the two functions share significant amount of computation. Therefore, in this work, we focus on the partitioned model and the mixed-partitioned model, and generalize them to a general partitioned model.

To explain the task models, we define a transition graph $G_k$ for each FSM $F_k$, where each transition is represented by a node and there is a directed edge between two nodes/transitions $\theta_i$ and $\theta_j$ when the source state of $\theta_i$ and $\theta_j$ is the same and the order of $\theta_i$ is lower than the order of $\theta_j$. A partition $P$ divides all the nodes in $G_k$ into a partition graph $G_k(P)$, where each partition set $p_i$ is represented by a node $v_i$, and a directed edge exists between two nodes $v_i$ and $v_j$ if at least one transition in $p_i$ has lower order than another transition in $p_j$. Fig. 3.1(b) shows a special partition graph $G_k(P_e)$ that is based on the event partition $P_e$, where the nodes/transitions are divided based on their events.

A multi-task implementation of the FSM $F_k$ consists of a partition $P$, where the transitions in each set $p_i$ is implemented by a task $\tau_i$. We assume a fixed priority scheduling among tasks, therefore a feasible task implementation needs to be consistent with the transition priorities, i.e. $\tau_i$ has higher priority than $\tau_j$ if a transition in $\tau_i$ has lower order than a transition in $\tau_j$. During execution, $\tau_i$ sends an inhibition signal to all lower priority tasks $\tau_j$ generated from the same FSM, if one of its transitions executes. When this happens, $\tau_j$ skips. It is easy to see that a task implementation has a feasible priority assignment only if its corresponding partition graph $G_k(P)$ is acyclic. For instance, there is no feasible task implementation based on the event partition $P_e$ in Fig. 3.1(b). In this case, transitions
\{\theta_1, \theta_2, \theta_3, \theta_4\} \text{ form a 4-cycle, where a task partition based on events } e_1 \text{ and } e_2 \text{ is infeasible.}

As the smallest possible cycles in event partitions, 4-cycles need to be addressed in any multi-task implementation.

**Partitioned model**  The partitioned model only applies to the FSMs where the event partition graph is acyclic. The multi-task implementation in the partitioned model is based on the event partition. Each task is executed at the period of the corresponding event. The tasks implementing the FSM are activated synchronously with offset = 0 and share a variable encoding the current state of the FSM. The task deadlines are assigned to ensure that any transition completes before the next set of inputs is processed. As shown in [22], the absolute deadline of a task instance (implementing transitions/actions) is the minimum value between the end of its period and the earliest activation time of higher priority tasks implementing transitions of the same FSM. This deadline guarantees that any task finishes its action execution and state update before a new task instance for the same FSM starts execution, and that there is no preemption between tasks from the same FSM.

**Mixed-partitioned model**  The partitioned model does not apply to the FSMs in which the event partition graph is cyclic. Therefore, a mixed-partitioned model is proposed in [22]. The idea is to identify all the transitions that belong to a cycle in the event partition graph and implement them in a single task \( \tau_b \), while the remaining transitions are implemented according to the partitioned model (as in Figure 3.1(c)). \( \tau_b \) will be assigned the highest priority and a period equal to the GCD of all the periods of the events that trigger transitions in \( \tau_b \).
As explained earlier, in Fig. 3.1, \{\theta_1, \theta_2, \theta_3, \theta_4\} form a 4-cycle that cannot be partitioned based on events. In the mixed-partitioned model, the four transitions are implemented in a single task \(\tau_1\) with highest priority and period 1\(ms\), while \(\theta_5\) is implemented in \(\tau_2\) with lower priority and period 3\(ms\). The deadlines of transitions \(\theta_1\) to \(\theta_4\) are 1\(ms\); the deadline of \(\theta_5\) is also 1\(ms\) because \(\tau_2\) has to finish before the activation of the higher priority task \(\tau_1\). In this case, the breakdown factor and the action extensibility are the same as in the single-task implementation.

**General partitioned model** The mixed-partitioned model proposed in [22] generates a particular task implementation, but many other multi-task implementations are possible, as long as the corresponding task partition graph is acyclic. As shown later in the motivating example, those multi-task implementations may provide better breakdown factor and action extensibility than the mixed-partitioned model. The *general partitioned model* is formally defined as follows.

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

Based on this definition, the illustration of general partitioned model of is shown in Fig.3.2. The FSM model is shown on the top and the task implementations based on
general partitioned model is shown on the bottom. All three implementations are valid. In
the task implementation, the dot/vertex denotes a transition in FSM, the arrow between
dots denotes the priority relations, i.e. transition \( \theta_1 \) has higher priority than transition \( \theta_2 \).
The task period is marked as blue.

The general partitioned model for Fig. 3.1 is simply an alternative of its mixed-
partitioned model implementation. By simply swapping the priorities of \( \tau_1 \) and \( \tau_2 \), resulting
in the implementation of Fig. 3.3. The periods of \( \tau_1 \) and \( \tau_2 \) are still 1\( ms \) and 3\( ms \). However,
the deadline of \( \theta_5 \) is now 3\( ms \) since \( \tau_2 \) has higher priority, which increases its extensibil-
ity. The deadlines of the task implementing the other four transitions is still 1\( ms \). The

Figure 3.2: The Implementations of General Partitioned Model
extensibility of $a_1$ to $a_4$ are the same as before. The computation of the extensibility of $a_5$ is more complicated – we need to consider the impact of an increase in $C_{a_5}$ on the four transitions mapped into $\tau_1$. As shown in Fig. 3.3, assuming the execution of $a_5$ starts at time $k$ (after $\theta_5$ is triggered by $e_2$), event $e_1$ may arrive at time $k + 1$ and enable $\theta_1$, with a deadline at $k + 2$. To ensure $\theta_1$ completes before $k + 2$, the execution time of $a_5$ can be at most $k + 2 - C_{a_1} - k = 1.6$ ($\theta_1$ is not activated until $\theta_5$ completes because $\tau_1$ has lower priority, therefore the functional correctness is preserved). It can be proved that this is the worst-case scenario for extending the execution time of $a_5$, and the extensibility of $a_5$ is $1.6/0.4 = 4$, larger than in the previous cases. The breakdown factor is still $1/0.5 = 2$.

The last task model improves the action extensibility but not the breakdown factor. A different general partition further improves on both metrics. We implement \{\theta_1, \theta_2, \theta_3\} in $\tau_1$, and \{\theta_4, \theta_5\} in $\tau_2$. $\tau_2$ is assigned a priority higher than $\tau_1$, and a period of 3ms.
\( \tau_1 \) has a period of 1ms. The extensibility of \( a_1, a_2, a_3 \) is still the same. The extensibility of \( a_5 \) is 4, the same as in the last implementation. The extensibility of \( a_4 \) increases from \( 1/0.5 = 2 \) to \( 1.6/0.5 = 3.2 \). This is computed by considering the case in which \( e_1 \) arrives 1ms after \( \theta_4 \) starts (the analysis is similar as for \( a_5 \)). Furthermore, the breakdown factor is also reached in this case – \( e_1 \) arrives 1ms after and both \( \theta_4 \) and \( \theta_1 \) need to complete within 2ms. The breakdown factor is therefore \( 2/(0.5 + 0.4) = 2.22 \) (all actions are scaled by the same factor). It can be proved that the system is schedulable with all tasks scaled by 2.22.

Note that the partitioned model and the mixed-partitioned model are clearly special cases of the general partitioned model. In the extreme case, we may implement each transition in the FSM with an individual task, although the overhead will be typically too high. Our goal is to explore various implementations based on general partitioned models to improve the breakdown factor and the action extensibility.

### 3.1.2 Metrics for FSM Task Implementations

In this section, we present methods to compute the quantitative metrics, i.e. breakdown factor and the action extensibility of any given task implementation based on the general partitioned model, for a set of synchronous FSMs implemented on a single processor. The algorithm to compute the metrics is proposed in our work in [76] and is presented here for convenience. We first present the exact formulation based on request bound function (rbf) and demand bound function (dbf) computation, given the high computation overhead using the exact formula, we also proposed approximated formulation for speeding up the computation of action extensibility.
a) Exact Formulation for Breakdown Factor and Action Extensibility

**Breakdown Factor**  Breakdown factor is defined as the maximum scaling factor applied to for all transitions that retains system schedulability, it can be formally defined as follows:

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

To compute the breakdown factor, we use a binary search to explore the range of $\lambda$. For each choice of $\lambda$, we check the schedulability of $T_{F_k}^\lambda$ based on the request bound function (rbf) and demand bound function (dbf) for each task $\tau_i \in T_{F_k}^\lambda$. The concepts of rbf and dbf are first introduced in [8] for the analysis of task graphs. In [71], algorithmic solutions are proposed to compute the rbf and dbf for synchronous FSMs, along with the condition for schedulability. Specifically, the rbf of a task $\tau_i$ during a time interval $\Delta = [s, f)$, denoted by $\tau_i.\text{rbf}(\Delta)$, is the maximum sum of the execution times by the actions that are implemented in $\tau_i$ and have their activation time within $\Delta$. The dbf of $\tau_i$ during $\Delta = [s, f)$, denoted by $\tau_i.\text{dbf}(\Delta)$, is the maximum sum of the execution times by the actions that are implemented in $\tau_i$ and have their activation time and deadline within $\Delta$. The schedulability of $\tau_i$ can be checked by the following condition, where $\pi_i$ is the priority of $\tau_i$ ([71]):

**Theorem 3** Task $\tau_i$ is schedulable if the following constraint is satisfied for all level-$\pi_i$
busy periods \([s, f]\):

\[
\forall t \in [s, f], \exists t' \in [s, t] \text{ such that } \tau_i.dbf[s, t] + \sum_{\tau_j \in HP_{\tau_i}} \tau_j.rbf[s, t'] \leq t' - s
\]  

(3.1)

where \(HP_{\tau_i}\) consists of tasks with higher priorities than \(\tau_i\).

The concept of level-\(\pi_i\) busy period is introduced in [41]. In a system with periodic trigger events, level-\(\pi_i\) busy periods start at the arrival points of the events within the hyperperiod that trigger tasks with priority higher than or equal to \(\pi_i\). For instance, assuming there are two events with periods \([2, 3]\), the start times of the busy periods to be considered are \([0, 2, 3, 4]\) – these are the set of \(s\) to be checked in (3.1). For the choices of \(t\) and \(t'\) to be checked in (3.1), we only need to consider the cases where \(t\) is the deadline of an instance of \(\tau_i\) and is within the length of the busy period (for which a bound is derived in [71]) and \(t'\) is at the arrival time of an event.

Leveraging Theorem 3, we propose Algorithm 1 for computing the breakdown factor of \(\mathcal{T_F}\). \(d_{\theta_i}\) is the shortest deadline for a transition \(\theta_i\) among all its activations. Let \(\tau_{\theta_i}\) denote the task that implements \(\theta_i\) (if \(\theta_i \in F_k\), then \(\tau_{\theta_i} \in T_{F_k}\)). For \(\tau_i \in T_{F_k}\), let \(HP_{\tau_i}^f\) denote the tasks that are in \(T_{F_k}\) (i.e. also implementing the transitions from \(F_k\)) and have higher priority than \(\tau_i\). \(\Theta\) is the set of all transitions in \(\mathcal{F}\), i.e. \(\Theta = \{\theta_i \mid \forall F_k \in \mathcal{F}, \theta_i \in F_k\}\).

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

36
\textbf{Algorithm 1} $\lambda^{\text{max}}_{\mathcal{T}_\mathcal{F}} = \text{compute\_breakdown\_factor}(\mathcal{F}, \mathcal{T}_\mathcal{F})$

1: $\forall \theta_i \in \Theta$, compute deadline $d_{\theta_i}$ as the GCD of the task periods from task set $\{\tau_{\theta_i}\} \cup HP_{\tau_{\theta_i}}$

2: $\lambda_{lb} = 0; \lambda_{ub} = \min\{d_{\theta_i} \mid \theta_i \in \Theta\}$

3: \textbf{while} $(\lambda_{ub} - \lambda_{lb} > \text{MAX\_ERROR})$ \textbf{do}

4: \hspace{1em} $\lambda = (\lambda_{ub} + \lambda_{lb})/2$

5: \hspace{1em} get $\mathcal{F}^\lambda$ and $\mathcal{T}_{\mathcal{F}^\lambda}$ by scaling $C_{a_i}$ by $\lambda$ for all action $a_i$

6: \hspace{1em} schedulable = \textbf{true}

7: \hspace{1em} \textbf{for all} $\tau_i$ in the system \textbf{do}

8: \hspace{2em} calculate start points $s$ for level-$\pi_i$ busy periods

9: \hspace{2em} calculate bound $f$ and choices of $t$, $t'$ for each busy period

10: \hspace{2em} \textbf{for all} $(s, t)$ pair for $\tau_i$ \textbf{do}

11: \hspace{3em} compute $\tau_i.dbf[s, t]$

12: \hspace{3em} for all $t'$, compute $\tau_j.rbf[s, t']$ and check Constraint 3.1

13: \hspace{3em} \textbf{if} Constraint 3.1 is not satisfied for any $t'$ \textbf{then}

14: \hspace{4em} schedulable = \textbf{false}; break out of the loops

15: \hspace{3em} \textbf{if} schedulable \textbf{then} $\lambda_{lb} = \lambda$ \textbf{else} $\lambda_{ub} = \lambda$

16: \hspace{1em} \textbf{return} $\lambda$
the same FSM.

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

**Definition 4** Given a set of synchronous FSMs $F = \{F_1, F_2, \ldots, F_m\}$ and its task implementation $T_F = \{T_{F_1}, T_{F_2}, \ldots, T_{F_m}\}$, the action extensibility $\beta_{a_i}$ of $a_i \in F_k$ is the largest scaling factor that, applied to $C_{a_i}$ (i.e. $C'_{a_i} = C_{a_i} \cdot \beta_{a_i}$) retains the schedulability of the task set.

A similar approach to the breakdown factor can be used for computing the action extensibility, i.e. a binary search on the value of $\beta_{a_i}$, checking the system schedulability for each candidate value.

**b) Approximating Action Extensibility**

**Algorithm 2** compute_action_extensibility_single($F$, $T_F$)

1: $\forall \theta_i \in F$, compute deadline $d_{\theta_i}$ as the GCD of the task periods from task set $\{\tau_\theta_i\} \cup HP_\theta_i$

2: for all $\theta_i \in F$ do
3: \hspace{1em} $slack^{\text{min}} = d_{\theta_i} - C_{a_i}$
4: \hspace{1em} for all $\theta_j$ s.t. $\tau_{\theta_j} \in LP_{\theta_i}$ do
5: \hspace{2em} $slack = d_{\theta_j} + \text{GCD}(\pi_{\theta_i}, \pi_{\theta_j}) - C_{a_i} - C_{a_j}$
6: \hspace{2em} for all $\tau_k \in (HP_{\theta_j} \setminus HP_{\theta_i} \setminus \{\tau_{\theta_i}\})$ do
7: \hspace{3em} $slack = slack - \max\{C_{a_m} | \mu(\theta_m) = \tau_k\}$
8: \hspace{2em} $slack^{\text{min}} = \min(slack^{\text{min}}, slack)$
9: \hspace{1em} $\beta_{a_i} = (slack^{\text{min}} + C_{a_i})/C_{a_i}$

38
Computing action extensibility using rbf and dbf based formula is very time consuming. If there is only one FSM on the processor and the largest action execution time is not larger than the GCD of the event periods, a more efficient algorithm (Algorithm 2) can be used to analyze the worst case scenario for action extensibility and directly compute its value with conservative approximations (not overly pessimistic in our tests). The main idea is to check the schedulability of the first instance of any lower priority task after the triggering of $a_i$. The motivating example shown in Fig. 3.3 shows the simplest case of such analysis.

### 3.1.3 FSM Task Implementation Optimization

Several algorithms can be designed to explore different task implementations with respect to the breakdown factor and the action extensibility. The brute force solution is to explore all possible task implementations based on the general partitioned model, and calculate the two metrics for each implementation using Algorithm 1 and 2. However, the complexity is likely to be too high for practical systems, where multiple FSMs are implemented on a single processor and each FSM consists of hundreds or thousands of states and transitions. Another option is to use randomized algorithm such as simulated annealing, which is much faster than brute force search and usually leads to high quality solutions, however it is still very time consuming for complex systems. We present an efficient heuristic that combines local optimization with global correctness checking and can be easily customized to optimize either of the two metrics (the use of different metrics may lead to different optimal solutions). The method was proposed in our work in [76] and is presented here for convenience.
First, to optimize the action extensibility, we need to extend the extensibility definition to a system level metric.

**Definition 5** Given a set of synchronous FSMs \( \mathcal{F} = \{F_1, F_2, \ldots F_m\} \) and its task implementation \( \mathcal{T}_F = \{T_{F_1}, T_{F_2}, \ldots T_{F_m}\} \) on a single processor, the system action extensibility \( B_{\mathcal{T}_F} \) is defined as the weighted average of the action extensibility of each action \( a_i \) in the system, i.e. \( B_{\mathcal{T}_F} = \sum_{a_i \in \bigcup_k F_k} w_i \cdot \beta_{a_i} \) where \( w_i \) is a pre-assigned weight that indicates the importance of the action and how likely it will be increased in future upgrades.

Other options include considering functional dependencies among actions and take into account the simultaneous update of a set of actions (as proposed in [75] for tasks) or to consider the minimum extensibility among all actions in the system. Our algorithm can be easily extended to handle other formulations like those.

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

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

a) Synthetic Examples

We use the TGFF tool [23] to generate random graphs and extend them to random FSM models (with added transitions, randomly selected periods, execution times and priorities) as our experiment sets. We compare the task implementations generated from our Algorithm 3 with the single-task implementation of the FSMs.

First, we compare the system action extensibility, for cases with a single FSM having from 5 to 25 states. The results are shown in Fig 3.4. The line labeled harmonic-fixed represents the cases where all event periods are harmonic and the transition priorities entirely depend on their events (the event partition is feasible). The line harmonic-50% represents the cases with harmonic periods and the transition evaluation orders depending on their events with a 50% probability (which might lead to 4-cycles). The other two lines...
are for non-harmonic event periods. Each data point in the figure is the average result of 20 random FSMs. For harmonic cases, our algorithm provides an improvement between 20% to 50% over the single-task implementation. For non-harmonic sets, the improvement increases to between $2X$ and $3X$. Non-harmonic sets allow more improvement since the deadlines in our multi-task implementations are typically much larger than in the single-task solution. When computing the action extensibility of our task implementation, we use Algorithm 2 since this is a single-FSM system. Action extensibility can be computed within 10 minutes for FSMs with 250 states.

Next, we compare the breakdown factor. Because of the timing complexity for computing the breakdown factor in Algorithm 1, we tested FSMs with size up to 12 states. The results are shown in Fig. 3.5, for the same sets of randomly generated FSMs used in the extensibility case. For non-harmonic cases, the improvement is between 20% and 40%,
while for harmonic cases it is around 10%. The improvement on the breakdown factor is less than the action extensibility, because it is limited by the action with the smallest timing slack, which is harder to improve.

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

b) Industrial Case Studies

![Figure 3.6: Action Extensibility Improvement of Industrial Case](image)

Figure 3.6: Action Extensibility Improvement of Industrial Case
We used a state machine model extracted from the internal logic of a real fuel injection control application to compare the action extensibility of task implementations in a case study with realistic complexity. After flattening all the hierarchical states and computing the product equivalent of series and parallel connection, the case-study model is a single FSM with 101 states and 446 transitions. Algorithm 3 provides a multi-task implementation with 6 tasks in 3 minutes, and Algorithm 2 computes the action extensibility in 1 minute. Fig. 3.6 illustrates the action extensibility improvement with different experiment configurations (priority assignment), it shows that the multi-task implementation has an improvement from 3.5X to 4.3X times over the single task implementation.

3.2 Single-Rate Synchronous Block Diagram Synthesis

As stated in [21], 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. 3.7 shows

Figure 3.7: Different task schedules result in different output latencies
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 3.2.3 and Section 3.2.4, besides task scheduling, different task generations may lead to different output latencies as well.

In this section, we present synthesis flow in which system timing (latency) is optimized and its tradeoff between other metrics are investigated. We also optimize for various software engineering metrics, including reusability, modularity, code size, etc. These metrics are defined and investigated in [43, 45, 44, 55] and will be discussed in following sections for convenience.

The overview of the proposed synthesis flow is shown in Figure 3.8. In this synthesis flow, tasks are generated and scheduled from synchronous block diagrams with different rates, whereas each SBD are assigned with a fixed rate (shown in upper the right of each SBD). TRCM, MRCT and integrated formulations drive the task synthesis and will be discussed in detail in the following sections.

3.2.1 System Models

The system model was presented in our work in [21] and is illustrated here for convenience.

**Task Generation Model.** During synthesis, a set of software tasks \( \Gamma^k = \{ \tau_1^k, \tau_2^k, \ldots, \tau_n^k \} \) is generated to implement the functionality of each \( \mathcal{S}^k \). Each task \( \tau_i^k \)
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 [43]). 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 pre-emption). 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. $r_{m}^{k}$ denotes the response time of task $\tau_{m}^{k}$, 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. $l_{y^k_j}$ represents the delay of output $y^k_j$ (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, $l_{y^k_j}$ is equal to the response time of the task to which $\hat{b}^k_j$ is mapped.

**Constraints and Objective Functions.** For each $y^k_i \in Y^k$, $l_{y^k_i}$ 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, $l_{y^k_i}$ should be less than or equal to the activation period (deadline) of the SBD to which it belongs, i.e. $l_{y^k_i} \leq T^k$. 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.

**a) 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.

$$f = \min_{y^k_i \in Y'} \{ \max \{ l_{y^k_i} \} \}$$

Another metric is to minimize the weighted sum of (a subset of) output latencies as in (3.3),
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.

$$f = \min \sum_{y^k_i \in Y'} \omega_{i,k} l_{y^k_i}$$  \hspace{1cm} (3.3)

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 (3.2).

$$f = \max \{ \min_{y^k_i \in Y'} \{ (T_k - l_{y^k_i}) \} \}$$  \hspace{1cm} (3.4)

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 (3.3).

$$f = \max \{ \sum_{y^k_i \in Y'} \omega_{i,k} (T_k - l_{y^k_i}) \}$$  \hspace{1cm} (3.5)

In this work, we focus on the timing metrics as in Equation (3.2) and (3.3) as representatives.

\textit{b) Modularity, reusability and code size.} The other three objectives are evaluated based on the definitions in [43, 44]. 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.

An illustrating example is shown in Fig. 3.9. 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 implementation has the optimal modularity and code size as no block is included in more than on tasks. However, the feedback connection at the top right of Fig. 3.9 is not allowed, leading to limitations in its reuse. In reality, there is no dependency between $o_1$ and $i_2$ 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) $c_1$, 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 $i_2$. At the bottom right of Fig. 3.9, the connection from $o_1$ to $i_2$ becomes possible even when $c_1$ and $c_2$ are considered as black-boxes since no loop exists. However, $b_1$ 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 $b_1$ is executed only once at each activation. Besides, the improvement of reusability sacrifice modularity comparing to single task implementation.

Reusability and modularity are metrics that apply to the selection of the set of functional components to be generated for each subsystem. These components may be considered in later stages (for example in a software supply chain) as the atomic units for composing the system design, the thread implementation and the scheduling model.

Next, we discuss the trade-off of different task implementation and then propose
Figure 3.9: Different task implementations result in different modularity, reusability and code size

algorithms for several SBD synthesis problems with different optimization metrics.

3.2.2 Tasks Implementations Trade-offs

We use the running example in Fig. 3.10 to illustrate different task generation schemes under different optimization objectives. The number in each block denotes its execution time.

Figure 3.10: An example of single rate SBD
a) 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. 3.10, if the model is implemented as a single task, 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 flexibility in selecting the optimal scheduling solution when multiple SBDs need to be scheduled as threads by a priority-based scheduler.

b) Modularity and Reusability

In [43], 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. 3.10, the approach generates two tasks as in Fig. 3.11: \( \tau_1 \) 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.

![Task implementations of Fig. 3.10 with best possible modularity and maximum reusability](image)

Figure 3.11: Task implementations of Fig. 3.10 with best possible modularity and maximum reusability
c) Modularity, Reusability and Code size

An approach is proposed in [44] to optimize modularity while achieving maximum reusability and minimum code size. For the example in Fig. 3.10, the approach generates three tasks as in Fig. 3.12: \( \tau_1 \) 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.

![Diagram showing task implementations of Fig. 3.10 with best possible modularity, minimal code size and maximum reusability](image)

Figure 3.12: Task implementations of Fig. 3.10 with best possible modularity, minimal code size and maximum reusability.

d) 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. 3.10, we assume the timing objective is the sum of output latencies as in Equation (3.3), with the weights set to 1. The approach proposed in this paper (as discussed next) generates the solution in Fig. 3.13: \( \tau_1 \) implements \( B \), \( \tau_2 \) for \( A, D \), \( \tau_3 \) for \( C, E, F, G \), 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.

![Figure 3.13: Task implementation of Fig. 3.10 for TRCM](image)

e) 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. 3.10, this will result in the same task set as in Fig. 3.12, 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).


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 3.2.5, 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 3.2.5 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 3.2.3 and 3.2.4.

Next, we introduce an algorithm to solve the TRCM problem with the sum of latencies metric (3.3), and its modification for the MRCT problem. We also discuss the solutions for the TRCM and MRCT problems for the max latency metric (3.2). The solutions were originally presented in our work in [21] and are summarized in following sections.

### 3.2.3 Synthesis for Timing, Reusability and Code Size, then Modularity

As stated in [21], for every output $y^k_i$, $Z(y^k_i) \subseteq B^k$ is its input zone set, containing the block $b$ that generates $y^k_i$ and the blocks belonging to the transitive closure of its input dependencies (all the blocks on which $b$ causally depends). For every $b^k_i$, we define $O(b^k_i) = \{y^k_j | b^k_i \in Z(y^k_j)\}$ (the set of all the outputs that have $b^k_i$ in their input zone) as its output set.

Our synthesis algorithm defines an execution order for all the tasks from the same SBD. $y^k_i$ is generated when all the tasks that include blocks from $Z(y^k_i)$ 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 $y^k_i$ is before $y^k_j$ in the output generation order, $I_t(y^k_i, y^k_j)$ is defined as the interval task set, which is the set of tasks scheduled after $y^k_i$, but before $y^k_j$ is generated. The blocks in $I_t(y^k_i, y^k_j)$ are the interval block set $I_b(y^k_i, y^k_j)$. $P_o(y^k_i)$ denotes the set of outputs that are before $y^k_i$ in the output generation order plus $y^k_i$ itself, and $P_b(y^k_i)$ denotes the set of blocks scheduled before $y^k_i$ 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 [44, 55], 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 6** 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., $P_b(y^k_i) = \bigcup_{y^k_j \in P_o(y^k_i)} Z(y^k_j)$.

**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 $y^k_j$ in $P_o(y^k_i)$ should be scheduled before $y^k_i$ can be generated. That is, $P_b(y^k_i) \supseteq \bigcup_{y^k_j \in P_o(y^k_i)} Z(y^k_j)$.

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 $b \notin \bigcup_{y^k_j \in P_o(y^k_i)} Z(y^k_j)$), 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 $P_o(y^k_i)$ 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 $P_b(y^k_i) = \bigcup_{y^k_j \in P_o(y^k_i)} Z(y^k_j)$. ■

**Theorem 7** 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. $b^k_i \in I_b(y^k_m, y^k_n)$ and $b^k_j \in I_b(y^k_m, y^k_n)$.

**Proof.** The theorem can be proved by contradiction, using Lemma 6. Assume the blocks $b^k_i$ and $b^k_j$ are not scheduled within the interval block set of two consecutive outputs, there must exist an output $y^k_l$ that is generated between the scheduling of the two blocks. Without loss of generality, we assume $b^k_i$ is scheduled before $y^k_l$ is generated, and $b^k_j$ is scheduled after $y^k_l$ is generated, i.e. $b^k_i \in P_b(y^k_l)$ and $b^k_j \notin P_b(y^k_l)$.

Based on Lemma 6, in any optimal solution, $P_b(y^k_l) = \bigcup_{y^k_j \in P_o(y^k_l)} Z(y^k_j)$. Therefore, we have $b^k_i \in \bigcup_{y^k_j \in P_o(y^k_l)} Z(y^k_j)$ and $b^k_j \notin \bigcup_{y^k_j \in P_o(y^k_l)} Z(y^k_j)$. This means that there exists $y^k_p \in P_o(y^k_l)$ such that $b^k_i \in Z(y^k_p)$ and $b^k_j \notin Z(y^k_p)$, which leads to $y^k_p \in O(b^k_i)$ and $y^k_p \notin O(b^k_j)$. 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 8** If $Succ(b^k_i) = \{b^k_j\}$, i.e. $b^k_j$ is the only successor of $b^k_i$, 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 $O(b_i^k) = O(b_j^k)$. Based on the Theorem 7, $b_i^k$ and $b_j^k$ 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 $\tau_n^k$ will not change since the only successor of $b_i^k$ is $b_j^k$. Similarly, the set of inputs on which $\tau_n^k$ depends will not be affected. Therefore, the move does not introduce false input-output dependencies through $\tau_n^k$. The move also will not introduce false input-output relation through $\tau_m^k$ (not needed if $b_j^k$ was its only block), since the set of outputs that depend on $\tau_m^k$ will not change and the set of inputs that $\tau_m^k$ 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 $b_i^k$ to the same task of $b_j^k$ preserves the optimality of the solution, we prove that there exists at least one optimum solution, in which $b_i^k$ and $b_j^k$ are in the same task. □

**Theorem 9** If $\text{Pred}(b_j^k) = \{b_i^k\}$, i.e. $b_i^k$ is the only predecessor of $b_j^k$, and $O(b_i^k) = O(b_j^k)$, and there is no input directly connected to $b_j^k$, then there exists at least one optimum solution of TCRM, in which $b_i^k$ and $b_j^k$ are in the same task.

**Proof.** The proof is similar as in Theorem 8. Assume there is an optimum solution with $b_i^k$ and $b_j^k$ in different tasks $\tau_m^k$ and $\tau_n^k$, respectively. We prove that moving $b_j^k$ from $\tau_n^k$ to $\tau_m^k$ produces another optimum solution.

Since $O(b_i^k) = O(b_j^k)$, based on the Theorem 7, $b_i^k$ and $b_j^k$ 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 $\tau^k_m$ 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 $\tau^k_m$. The move also will not introduce false input-output dependencies through $\tau^k_n$, since the set of outputs that depend on $\tau^k_n$ and the set of inputs on which $\tau^k_n$ depend on will not change (not needed if $b^k_j$ was the only block of original $\tau^k_n$). 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 $b^k_j$ to the same task of $b^k_i$ 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 8 and 9, 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 TCRM. 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 $O(b^k_i)$ of every block $b^k_i$. Next, the blocks that satisfy the conditions of Theorems 8 and 9 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 $u_{b^k_i,y^k_j}$ represents whether $b^k_i$ is executed before $y^k_j$ is generated, and $v_{y^h_k,y^k_j}$ represents whether $y^h_k$ is generated before $y^k_j$. 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 $p_{s^m,s^k}$ denotes whether the tasks from $S^m$ have higher priority than those from $S^k$. $C_{b^k_i}$ denotes the execution time of block $b^k_i$. $C_{S^k}$ 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 (3.6)) is the weighted sum of latencies as in (3.3). Constraint (3.7) sets the values for those $u_{b^k_i,y^k_j}$ that can be statically determined from the diagram. Constraint (3.8) enforces the mutual consistency of the $u$ and $v$ variables, and (3.9) enforces the internal consistency of the $v$ values (anti-reflexivity and transitivity). Constraint (3.10) 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 (3.11–3.13) uses a set of additional integer variables and the standard “big M” formulation in mixed-integer programming [48]. $z_{m,j,k}$ is the maximum (integer) number of possible activations of the tasks generated from $S^m$ within $l_{y^k_j}$. $i_{m,j,k}$ is the maximum number of activations that are actually generating interferences (when the tasks
generated from $\mathcal{S}^m$ have higher priority). Finally, the constraints in (3.14) enforce the anti-reflexive and transitive properties of the priority assignment. Constraint (3.15) encodes the timing constraint of the synchronous assumption.

\[
\min \sum_{y^k_j \in Y'} \omega_{j,k} \times l^k_j \tag{3.6}
\]

\[
u^k_{b_i^k, b_j^k} = 1 \quad \forall b_i^k \in Z(y_j^k) \tag{3.7}
\]

\[
u^k_{b_i^k, y_j^k} \geq \nu^k_{b_i^k, y_h^k} + \nu^k_{b_h^k, y_j^k} - 1 \tag{3.8}
\]

\[
u^k_{b_i^k, y_j^k} + \nu^k_{y_j^k, y_h^k} = 1; \quad \nu^k_{b_i^k, y_j^k} \geq \nu^k_{b_i^k, y_h^k} + \nu^k_{y_h^k, y_j^k} - 1 \tag{3.9}
\]

\[
l_j^k = \sum_i C_{b_i^k} \times u^k_i \times y_j^k + \sum_m C^m_i \times \nu^k_{m,j,k} \tag{3.10}
\]

\[
0 \leq \nu^k_{m,j,k} \leq z^k_{m,j,k} \tag{3.11}
\]

\[
z^k_{m,j,k} - M \times (1 - p^s_{m,k}) \leq \nu^k_{m,j,k} \leq M \times p^s_{m,k} \tag{3.12}
\]

\[
0 \leq z^k_{m,j,k} - \frac{l^k_j}{T_m} < 1 \tag{3.13}
\]

\[
p^s_{m,k} + p^s_{k,s^m} = 1; \quad p^s_{m,k} \geq p^s_{s^m,s^n} + p^s_{s^n,s^k} - 1 \tag{3.14}
\]

\[
l^k_j \leq T_k \tag{3.15}
\]

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. \( P_b(y_i) \)) and the interval block set \( I_b(y_m, y_i) \) for any
two consecutive outputs \( y_m \) and \( y_i \) (in the generation order). In our MILP formulation, we
define \( G_i \equiv I_b(y_m, y_i) \) (if \( y_i \) is the first output, \( G_i \equiv P_b(y_i) \)). 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.

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

The variable \( h_{\tau_{r,m}, \tau_{s,n}} \) represents whether \( \tau_{s,n} \) 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

$$\min \sum_{r,m} Z_{\tau_r,m} \quad (3.16)$$

$$Z_{\tau_r,m} \geq f_{b_r,i,\tau_r,m} \quad \forall i \quad (3.17)$$

$$\sum_m f_{b_r,i,\tau_r,m} = 1 \quad \forall r \neq X,Y \quad (3.18)$$

$$f_{b_r,i,\tau_s,m} = 0 \quad \forall r \neq s \quad (3.19)$$

$$f_{b_r,i,\tau_r,m} = 1 \quad \forall r = X,Y \land m = i \quad (3.20)$$

$$f_{b_r,i,\tau_r,m} = 0 \quad \forall r = X,Y \land m \neq i \quad (3.21)$$

$$h_{\tau_r,m,\tau_s,n} \geq f_{b_r,i,\tau_r,m} + f_{b_s,j,\tau_s,n} + Q_{b_r,i,b_s,j} - 2 \quad (3.22)$$

$$h_{\tau_r,m,\tau_s,n} \geq h_{\tau_r,m,\tau_r,l} + h_{\tau_l,l,\tau_s,n} - 1 \quad (3.23)$$

$$h_{\tau_r,m,\tau_s,n} + h_{\tau_s,n,\tau_r,m} \leq 1 \quad (3.24)$$

$$h_{\tau_r,m,\tau_s,n} = Q_{b_r,m,b_s,n} \quad \forall r = X \land s = Y \quad (3.25)$$

The constraints on the mapping of blocks to tasks are (3.18–3.21). Constraints (3.22–3.24) define the causality relations among tasks, and (3.25) 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 4 provides an alternative solution for modularity optimization.
Algorithm 4 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 $\mathit{change}$ do

3: \hspace{1em} \textbf{while} $\exists b_i \in G_k, b_j \in G_k$, s.t. $\mathit{Pred}(b_j) = \{b_i\}$ and there is no input directly connected to $b_j$ do

4: \hspace{2em} \textbf{if} schedulable_after_merge($b_i, b_j$) \textbf{then}

5: \hspace{3em} merge_blocks($b_i, b_j$); merge_tasks($\tau_i, \tau_j$).

6: \hspace{2em} \textbf{while} $\exists b_i \in G_k, b_j \in G_k$, s.t. $\mathit{Succ}(b_i) = \{b_j\}$ and there is no output directly connected to $b_i$ do

7: \hspace{3em} \textbf{if} schedulable_after_merge($b_i, b_j$) \textbf{then}

8: \hspace{4em} merge_blocks($b_i, b_j$); merge_tasks($\tau_i, \tau_j$).

9: \hspace{2em} \textbf{change} = false

10: \hspace{2em} \textbf{if} $\exists b_i \in G_k, b_j \in G_k$, s.t. max_reusability_after_merge($b_i, b_j$) and schedulable_after_merge($b_i, b_j$) \textbf{then}

11: \hspace{3em} merge_blocks($b_i, b_j$); merge_tasks($\tau_i, \tau_j$); $\mathit{change} = true$.

12: \textbf{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 9, 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 similar merge rule as Theorem 8 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 9 and 8 are checked again.

The above three-step algorithm is also applicable to the cases in which minimizing maximum latency (3.2) 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 (3.6) with the formulation.

$$
\begin{align*}
\left\{\begin{array}{c}
\text{min} & l_{\text{max}} \\
\text{s.t.} & l_{y_{ij}^k} \leq l_{\text{max}} & \forall y_{ij}^k \in Y'
\end{array}\right.
\end{align*}
$$

(3.26)

### 3.2.4 Synthesis for Modularity, Reusability and Code Size, then Timing

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 3.2.3), as long as no false input-output dependency is introduced and the system is schedulable. In [44], 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.

### 3.2.5 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 \( S^k \). \( \tau_{km} \) denotes the \( m \)-th task generated for SBD \( k \). \( f_{b^k_i, \tau_{km}} \) indicates whether \( b^k_i \) is mapped to \( \tau_{km} \).

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. \( b^k_i \) 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 \( |B^k| + |X^k| < i \leq |B^k| + |X^k| + |Y^k| \). \( \tau_{km} \) 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. \( b^k_m \)) when \( |B^k| < m \leq |B^k| + |X^k| + |Y^k| \).

\( Z_{\tau_{km}} \) 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 \( C \equiv |B^k| + |X^k| \).
and $\mathcal{D} \equiv |B^k| + |X^k| + |Y^k|$. The MILP formulation is as follows.

\begin{itemize}
  \item \textbf{a) Allocation constraints:}
    \begin{align}
      \sum_{1 \leq m \leq |B^k|} f_{b^k_i, r^k_m} &= 1 \quad \forall 1 \leq i \leq |B^k| \quad (3.27) \\
      f_{b^k_i, r^k_m} &= 1 \quad \forall |B^k| < i \leq \mathcal{D} \quad (3.28) \\
      f_{b^k_i, r^k_j} &= 0 \quad \forall i \neq j, |B^k| < j \leq \mathcal{D} \quad (3.29)
    \end{align}

  Constraints (3.27) to (3.29) enforce the mapping from blocks to tasks. In particular, constraint (3.27) defines the mapping from original (internal) blocks to tasks, and constraint (3.28) defines the mapping from virtual input and output blocks to their corresponding dedicated tasks.

  \item \textbf{b) Reusability constraints:}
    \begin{align}
      h_{r^k_m, r^k_n} &\geq f_{b^k_i, r^k_m} + f_{b^k_i, r^k_n} + Q_{b^k_i, b^k_j} - 2 \quad (3.30) \\
      h_{r^k_m, r^k_n} &\geq h_{r^k_m, r^k_j} + h_{r^k_i, r^k_j} - 1 \quad (3.31) \\
      h_{r^k_m, r^k_n} + h_{r^k_n, r^k_m} &\leq 1 \quad (3.32) \\
      h_{r^k_m, r^k_n} &= Q_{b^k_m, b^k_n} \quad \forall |B^k| < m \leq \mathcal{C} < n \leq \mathcal{D} \quad (3.33)
    \end{align}

  Constraints (3.30) to (3.32) define the causality relations among tasks, based on the causality relations among the blocks and the block-to-task mapping. Constraint (3.33) 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.
\end{itemize}
c) Timing constraints:

\[ c_{\tau_k} = \sum_i C_{b_i} \times f_{y_i, \tau_k} \]  

(3.34)

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

(3.35)

\[ x_{i,n,m,k} \leq f_{y_i, r_{\tau_k}} \]  

(3.36)

\[ x_{i,n,m,k} \leq u_{r_{\tau_m}} \]  

(3.37)

\[ f_{y_i, r_{\tau_k}} + u_{r_{\tau_m}, r_{\tau_k}} - 1 \leq x_{i,n,m,k} \]  

(3.38)

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

(3.39)

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

(3.40)

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

(3.41)

\[ l_{y_j} = \sum_m v_{j,m,k} \]  

(3.42)

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

(3.43)

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

(3.44)

\[ l_{y_j} \leq T_k \]  

(3.45)

\[ p_{s^i,s^k} + p_{s^k,s^i} = 1 \]  

(3.46)

\[ p_{s^i,s^k} \geq p_{s^i,s^j} + p_{s^j,s^k} - 1 \]  

(3.47)

\[ u_{r_{\tau_m}, r_{\tau_k}} \geq h_{r_{\tau_m}, r_{\tau_k}} \]  

(3.48)

\[ u_{r_{\tau_m}, r_{\tau_k}} + u_{r_{\tau_k}, r_{\tau_m}} = 1 \]  

(3.49)

Formula (3.34) to (3.41) represent the computation of task response times. On the right hand side of (3.35), the first term \( c_{\tau_m} \) represents the execution time of task \( \tau_k \) (computed
in (3.34)), the second term represents the total execution time of tasks from the same SBD and scheduled before $\tau_{m}^{k}$, and the third term represents the time $\tau_{m}^{k}$ needs to wait for higher priority tasks from other SBDs. The second term and the formula (3.36) to (3.38) are linearization of $\sum_{n} c_{n} \times u_{n} \tau_{m}^{k}$. The third term and the formula (3.39) to (3.41) are linearization of $\sum_{i} C_{s} \times p_{s} \times \lceil \tau_{m}^{k} / T_{i} \rceil$, using the “big M” formulation [48].

Constraints (3.42) to (3.44) relate the output latencies to the corresponding task response times. Constraint (3.45) enforces that output latencies are not greater than the activation period of the corresponding SBD; (3.46) and (3.47) enforce the anti-reflexive and transitive properties of the priority assignment; (3.48) and (3.49) set the scheduling among tasks from the same SBD, and ensure that it is consistent with the causality relation among tasks.

In below, formula (3.50) to (3.53) define a general multi-objective optimization problem [46] with respect to the timing metric $F$ and the modularity metric $G$. In general, there is a tradeoff between modularity and timing.

d) Objectives:

$$Z_{\tau_{m}^{k}} \geq f_{b_{i}^{k}} \forall i$$  (3.50)

$$G = \sum_{m,k} Z_{\tau_{m}^{k}}$$  (3.51)

$$F = \sum_{y_{j} \in Y'} \omega_{j,k} \times l_{y_{j}^{k}}$$  (3.52)

$$\min \left[ F, G \right]^{T}$$  (3.53)

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:

\[ G \leq G_{\text{MAX}} \]  
\[ \min \mathcal{F} \]  

where \( G_{\text{MAX}} \) is the upper bound of the number of tasks. By varying \( G_{\text{MAX}} \), 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 \( F^k \) and \( G^k \) defined as the timing and modularity of each SBD and a multi-objective optimization problem defined as the follows.

\[ \min [F^1, F^2, ..., F^{|S|}, G^1, G^2, ..., G^{|S|}]^T \]  

3.2.6 Experimental Results

As stated in [21], 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) Overview of Test Cases

**Industrial examples.** The first example is a portion of an automotive fuel injection system [51], 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 [50], 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).

The fuel injection example is shown in Fig. 3.14. The execution time of each block (in $\mu$s) is shown in the parenthesis. The period is 1000$ms$.

Figure 3.14: Fuel injection example
**Synthetic examples.** We also use TGFF [24] 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 3.1 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 (3.3), where we let \( Y' \) include all outputs and the weights are 1; and the maximum latency (3.2), 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. For both timing metrics, TRCM clearly provides better timing than MRCT, at the expense of worse modularity.

<table>
<thead>
<tr>
<th>Example</th>
<th>Sum of latencies</th>
<th>Max latency</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>S.L.</td>
<td>#T</td>
</tr>
<tr>
<td>Fuel Inj.</td>
<td>2991</td>
<td>13</td>
</tr>
<tr>
<td>Robotics</td>
<td>805</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3.1: Results of TRCM and MRCT for industrial case studies under two timing optimization metrics

**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 optimiza-
tion. Fig. 3.15 and 3.16 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).

Figure 3.15: Timing metric comparison between TRCM and MRCT for single-SBD synthetic examples

*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 MRCT results for the two timing metrics. The results are shown in Fig. 3.17 and Fig. 3.18.

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 [44].

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
Figure 3.17: Timing metric comparison between TRCM and MRCT for multi-SBD synthetic examples

TRCM and MRCT always provide maximum reusability. 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 [44] 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 [44]: we use a topological sort to determine the order among tasks gener-
Figure 3.18: Modularity comparison between TRCM and MRCT for multi-SBD synthetic examples

ated 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.
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 3.2 shows the runtimes for our synthetic multi-SBD examples (3 SBDs, each with 10 to 50 blocks).

<table>
<thead>
<tr>
<th>SBD sizes</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime (seconds)</td>
<td>1</td>
<td>8</td>
<td>55</td>
<td>148</td>
<td>535</td>
</tr>
</tbody>
</table>

Table 3.2: Average runtime of Step 2 timing optimization in TRCM for synthetic multi-SBD examples

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 3.3 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 3.3: 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.

### 3.3 Multi-Rate Synchronous Block Diagram Synthesis: For Automotive Systems

Many automotive controls are today specified and realized using Synchronous Reactive (SR) models such as those modeled with 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. The block diagram in those models are in general multi-rated, because of the physical limitations (e.g. sensors operates in different rates) of the systems.
Fig. 3.19 shows the software supply chain in automotive industry, following AUTOSAR standard. 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.

Many current approaches treat runnable synthesis and task synthesis as isolated steps and timing are overlook in runnable synthesis step, which result in high design effort and error-prone implementations. In this section, we present the integrated synthesis flow for multi-rate synchronous block diagram for automotive systems. We treat timing as our first class citizen and timing is optimized throughout the whole process. In the meantime, we also optimize for modularity, reusability, and code size as in Section 3.2. The proposed synthesis flow is shown in Fig. 3.20. At the runnable synthesis stage, we propose top-down and bottom-up based heuristics to generate runnables, with design considerations including timing (schedulability), modularity, reusability and code size. At the task synthesis stage, we propose greedy heuristic and simulated annealing based algorithms to generate task implementations on multicore platforms, with design considerations including schedulability and memory usage. The detailed synthesis flow will be discussed in the following sections.

3.3.1 From Multi-Rate SBD to AUTOSAR Runnables

The first step of the synthesis flow is to synthesize multi-rate SR models to runnables. We reuse the definition in Section 2.1.3 and Section 2.2 for SBD model and runnable model respectively.

a) FETA for the Specification of Activations and Execution Time Requests:

At runnable level, the underlying hardware, task mapping and scheduling are all unknown, it’s difficult to directly analyze schedulability at runnable level. In this work, we
propose to use FETA to capture the timing behavior at runnable level, FETA formulation is further used to capture timing on task level. Based on FETA, a quantitative metric expressing the schedulability requirements is proposed. The FETA formulation is proposed in our work in [19] and is presented here again in more details.

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) [45] 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 $A$ is a pair $A = (\theta, S)$ where $\theta$ 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 $\delta : v_{i-1} \rightarrow v_i$ is a link or transition, with $v_n \rightarrow v_0 \in \delta$, i.e. the FTA is a cycle. The cycle period of the FTA is $\Theta = \theta \cdot n$.

**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 is shown in Fig. 3.21 with the execution time requirements indicated with an FETA notation. The runnable implementation on the bottom is generated from the SBD model on the top. In this figure, the runnable inherit the dependencies from original block diagram, the number on the upper left corner denotes the period of block/runnable, and the runnable period is the GCD of periods of blocks that mapped to it, for instance, period of runnable \( r_1 \) is essentially GCD of 2, 5. Runnable name and the blocks mapped to it is denoted on the upper left corner. For instance, block \( A \), \( B \) and \( C \) are mapped to runnable \( r_1 \). In the FETA notation in the runnable implementation, each vertex denotes an activation and the double circle denotes an execution. The distance between adjacent vertices is the period of the runnable (e.g. 1 for \( r_1 \)), which essentially means that it will proceed by 1 vertex every runnable period cycle. The number below the vertex denotes the execution request at that vertex (marked by row “W”), the corresponding executing blocks are denoted in row “Blk”, and the number inside the circle denotes the time instance. Taking runnable \( r_1 \) as example, at time 0, the execution request is 0.7, because block \( A \), \( B \) and \( C \) will be executed at the same time. At time 1, nothing will be executed because period of block \( A \), \( B \) and \( C \) is 2, 5, 4 respectively. At time 2, only block \( A \) and \( C \) will be execution therefore the execution request at time 2 is 0.3, the worst case execution time of block \( A \).

The FETA specification for a runnable can be easily constructed from the timing specifications \((t, \gamma)\) 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.

Note that although we focus on synchronous block diagram model in this section, the FETA formulation is not constrained to SBD model, it can be easily extended to synchronous FSM model. Therefore, FETA can also be adapted for the timing analysis to our
work in Section 3.1. The adaption of FETA on FSM model is shown in Fig. 3.22. The FSM is triggered by events with different period, each transition takes certain worst-case execution time to finish, when mapping all the transitions into the same runnable/task, the activation and execution request can be captured by FETA shown in the bottom of Fig. 3.22.

![Figure 3.22: Using FETA on synchronous FSM model](image)

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

For the *oversampling* operator, the structure of $A'$ is $S' = (V', v_0', F', \delta')$, with $\|V\| = k \times n$. For each $v_i \in V$, with $i = 0, \ldots, n$, there are $k$ 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 $j = 1, \ldots, k - 1$. 

86
Finally $v_j^i \rightarrow v_{j+1}^i \in \delta'$ for all $j = 0, \ldots, k \cdot n - 1$ and $v_{k \cdot n}^i \rightarrow v_0^i \in \delta'$.

The result of the sum operation $A = B \oplus C$ is $S^A = (V^A, v_0^A, F^A, \delta^A)$ with $\|V^A\| = n^A = \text{lcm}(n^b, n^C)$ and $v_j^A \rightarrow v_{j+1}^A \in \delta^A$ for all $j = 0, \ldots, n^A - 1$ and $v_{n^A}^A \rightarrow v_0^A \in \delta^A$. For each $v_i^A \in V^A$, there are two nodes $v_j^B \in B$ and $v_k^C \in C$ 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 $v_j^B \in F^B \lor v_k^C \in F^C$.

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 $v_j^B \in F^B \land v_k^C \in F^C$, then $CE_A = CE_B \cup CE_C$. For unconditional executions, it is as follows (assuming the runnable index matches the FETA denomination):

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

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

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

b) Objectives and Constraints

As stated in [19], a sample multi-rate SBD model is shown in Fig. 3.23 and is used as running example to illustrate the design metrics and trade-offs. In this example, the
number on the right of each block denotes block period and the number on the bottom left denotes WCET.

![Diagram showing multi-rate SBD]

**Figure 3.23: An example of multi-rate SBD**

During runnable synthesis, 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. Note that the concepts of modularity and reusability are defined the same as in Section 3.2, except the definition in this section it is on runnable level. We introduce them again in this section from runnable’s view.
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.

For the running example in Fig. 3.23, if we simply map all blocks into a runnable, the feedback connection between output $o_1$ and input $i_5$, $o_2$ and $i_1$, etc. are not allowed, leading to limitation in its reuse. In comparison, the runnable implementation in Fig. 3.26 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 $F_k$ is further characterized by its local utilization $u_i = c_{i0}/\rho_i$, where $\rho_i = \theta_k \cdot d_i$ 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 $\theta_k$ is the period of $F_k$ as defined earlier).

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

$$U_l = \sum_{r_k \in R} u^\max_k \text{ where } u^\max_k = \max_v(u^v_k) \quad (3.57)$$
The component utilization $U_c$ is computed as:

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

(3.58)

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. $\alpha_u$ is used to estimate the schedulability of the runnable implementation (smaller $\alpha_u$ indicates better schedulability). The following theorem helps identify the runnable generation with maximum schedulability.

**Theorem 10** The alpha ratio $\alpha_u = \frac{U_l}{U_c}$ for any runnable implementation $R$ 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 $F_k$ of a runnable, let $d'$ be the path length from the vertex at which all the blocks in $F_k$ 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 $\alpha_u = 1$.

**Trade-Offs Illustrations:** Based on the discussion on the constraints and objectives, it is obvious that there are trade-offs among the design metrics. In general, better timing (abstract schedulability) will result in worse modularity. Three possible runnable
implementation from the model in Fig. 3.23 will be compared and the trade-offs are discussed below:

Fig. 3.24 shows the runnable synthesis results with optimal modularity in which only one runnable is generated. In this example, reusability is violated multiple times, e.g. there is false input-output dependency between \( i_1 \) and \( o_3 \). The runnable itself is not schedulable, because of multiple deadline violations, e.g. the first vertex takes 2.45ms to finish which exceeds 2ms deadline, and the fifth vertex exceeds 1ms deadline as it takes 1.45ms to finish.

![Figure 3.24: Runnable Synthesis with Optimal Modularity](image)

Optimal reusability and better schedulability can be achieved in Fig. 3.25. In this example, there is no false input-output dependencies therefore reusability is optimal. However the modularity is getting worse compared to previous example. The runnable itself has no timing violations, but the vertex at the red arrow has tight timing budget (i.e. execution takes 0.8ms and the deadline is 1ms), which may result in unschedulable system after task generations (i.e. considering preemption from higher priority tasks).

Better schedulability can be achieved in Fig. 3.26, again the reusability is optimal, but modularity is getting even worse. However, the timing budgets for all runnables are sufficient and this runnable synthesis solution may result in better schedulability in the
Figure 3.25: Runnable Synthesis with Optimal Reusability and Tight Timing

following task synthesis stage.

Figure 3.26: Runnable Synthesis with Optimal Reusability and Relaxed Timing

c) 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). The two methods were proposed in our work in [19] and is presented in this section for convenience. The top-down method starts with a runnable generation with the maximum modularity and gradually decompose the functions to improve the schedulability, which is measured by the alpha ratio defined in Section 3.3.1.b. The bottom-up method starts with a function partitioning with the maximum schedulability, and gradually merge the functions to improve the modularity. Both methods provide maximum reusability. Fig. 3.27 shows the overview of the top-down and bottom-up approach. The details of the two methods are introduced in below.

**Top Down Approach**

![Top Down Approach Diagram](image)

**Bottom Up Approach**

![Bottom Up Approach Diagram](image)

Figure 3.27: Runnable Synthesis Flow for Top-down and Bottom-up approaches

**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 [21]). 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 $B$ 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. $C_{bi}$ and $C_{rm}$ denote the component to which $b_i$ and $r_m$ belong. The binary variable $g_{bi,r_m}$ represents whether the block $b_i$ is mapped to runnable $r_m$, and $a_{bi,bj}$ denotes whether $b_i$ and $b_i$ are mapped to the same runnable. The variable $h_{rm, rn}$ captures the causality relation between runnables (if 1 then $r_m$ must be executed before $r_n$) and the parameter $Q_{bi,bj}$ records the causality relation between blocks and can be computed by using a BFS (Breath-first search) before running the MILP optimization. $Z_{rm}$ indicates whether there is at least a block mapped to runnable $r_m$, and therefore $\sum m Z_{rm}$ represents the number of runnables. The constraints on block-to-runnable mapping are (3.61–3.64). Constraints (3.65–3.67) define the causality relations among tasks, and (3.68) requires that the block-to-runnable mapping does not introduce any false input-output dependencies.

\[
\min \sum m Z_{rm} \tag{3.59}
\]

\[
Z_{rm} \geq g_{bi,r_m} \ \forall b_i, r_m \in B, R \tag{3.60}
\]
\[
\sum_{m} g_{b_i,r_m} = 1 \quad \forall b_i, r_m \in B, R 
\] (3.61)

\[
g_{b_i,r_m} = 0 \quad \forall C_{b_i} \neq C_{r_m} 
\] (3.62)

\[
g_{b_i,r_m} = 1 \quad \forall b_i, r_m \in X, Y \land m = i 
\] (3.63)

\[
g_{b_i,r_m} = 0 \quad \forall b_i, r_m \in X, Y \land m \neq i 
\] (3.64)

\[
h_{r_m,r_n} \geq g_{b_i,r_m} + g_{b_j,r_n} + Q_{b_i,b_j} - 2 
\] (3.65)

\[
h_{r_m,r_n} \geq h_{r_m,r_i} + h_{r_i,r_n} - 1 
\] (3.66)

\[
h_{r_m,r_n} + h_{r_n,r_m} \leq 1 
\] (3.67)

\[
h_{r_m,r_n} = Q_{b_m,b_n} \quad \forall b_m, r_m \in X \land b_n, r_n \in Y 
\] (3.68)

Starting from the solution obtained from the MILP formulation, the heuristic Algorithm 5 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 7 to 24). 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 \( \alpha \)). 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 \( \alpha \).

Algorithm 5 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.
Algorithm 5 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: $\alpha_u = \text{ComputeAlpha}()$

4: while $\alpha_u > \alpha_u' \lor N < N'$ 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: $r_n = \Phi, \text{found} = \text{false}, T_n = 0$

8: for all $b_i \in r_m$ in order do

9: if $\text{found} = \text{false}$ then

10: $r_n = r_n \cup b_i$

11: $p_t = \text{decompose}(r_m, r_n)$

12: if $\text{existsCycle}(p_t)$ then

13: $r_n = r_n - b_i$

14: else

15: $p_b = p_t, \text{found} = \text{true}, T_n = T_{b_i}$

16: else

17: if $T_{b_i} = T_n$ then

18: $r_n = r_n \cup b_i$

19: $p_t = \text{decompose}(r_m, r_n)$

20: if $\text{existsCycle}(p_t)$ then

21: $r_n = r_n - b_i$

22: else

23: $p_b = p_t$

24: $p = p_b, \alpha_u = \text{ComputeAlpha}(), N = N + 1$

25: return $p$
**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 10, 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

**Algorithm 6** 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: $U_l = \text{compLocalUtil}(p)$
4: $\alpha_u = U_l / U_c$
5: while $N > N' \land \alpha_u < \alpha'_u$ do
6: for all $r_m, r_n \in p$ do
7: $p_t = \text{merge}(r_m, r_n)$
8: if isSched($p_t$) then
9: $U_{pl} = \text{compLocalUtil}(p_t)$
10: if $\Delta U > U_{pl} - U_l$ then
11: $p_b = p_t, \Delta U = U_{pl} - U_l$
12: if $\exists p_b$ then
13: $p = p_b, \text{update}(\alpha_u), N = N - 1$
14: else
15: return $p$
16: return $p$
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 (3.69–3.72) on block periods to ensure that all blocks in a runnable have the same period.

\[
\begin{align*}
  a_{b_i,b_j} &\geq g_{b_i,r_m} + g_{b_j,r_m} - 1, \quad \forall b_i, b_j, r_m \in \mathcal{B} \tag{3.69} \\
  a_{b_i,b_j} &\leq g_{b_i,r_m} - g_{b_j,r_m} + 1, \quad \forall b_i, b_j, r_m \in \mathcal{B} \tag{3.70} \\
  a_{b_i,b_j} &\leq g_{b_j,r_m} - g_{b_i,r_m} + 1, \quad \forall b_i, b_j, r_m \in \mathcal{B} \tag{3.71} \\
  a_{b_i,b_j} &= 0, \quad \forall T_{b_i} \neq T_{b_j} \tag{3.72}
\end{align*}
\]

Starting from the MILP solution, the heuristic Algorithm 6 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.

### 3.3.2 From AUTOSAR Runnables to Tasks Implementations

The runnable synthesis in Section 3.3.1 generates runnables while optimizing schedulability, modularity, reusability and code size as in [19]. 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 $\mathcal{T}^p = \{\tau^p_1, \tau^p_2, \ldots, \tau^p_m\}$. The core index is dropped from tasks when not required. Each task $\tau_j$ has period $\theta_j$, activation offset $\phi_j$, and is scheduled according to its priority $\pi_j$ 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 $\tau_j$ is denoted as $w_j$. A mapping relation $\mathcal{M}(\tau_j, r^j_s, k)$ defines the execution of runnable $r^j_s$ with order $k$ (in a sequence of runnable calls) inside the code of task $\tau_j$ (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 $\tau_i$ 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 in Task Synthesis

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 [51]). 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 $\phi_i$ of a task $\tau_i$ is the maximum among the response times of all the predecessor runnables that have a successor in $\tau_i$. 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 [28]). 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 $\tau_i$ 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 $\Theta_{c_i} = \text{lcm}(\Theta_i)$. This least common multiple is the cycle period for the entire set of tasks. For each FETA $\mathcal{F}_i$ (or task
\(\tau_i\), we denote the set of activation times in the scheduling period plus the largest offset \(\Theta + \max_k(\phi_k)\) as \(\{a_i(1), a_i(2), \ldots, a_i(p)\}\). When considering the task offset, \(\phi_i\) is added to all the activation times \(a'_i(k) = a_i(k) + \phi_i\). At this point, leveraging the analysis in [28], the response time for a generic request in \(a'_i(k)\) of execution in the FETA of \(\tau_i\) is performed by considering all the busy periods of level \(\pi_i\) 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 [52] and the allocation heuristics in [68], [4]. 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 7 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
Algorithm 7 Greedy heuristic algorithm for task synthesis

1: $\mathcal{G}$ graph of all runnables with execution order dependencies

2: $\mathcal{L} = \emptyset$ 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 $Rc_i = \emptyset$, $Tc_i = \emptyset$ 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: $r_i = \text{GetFirst}(\mathcal{L})$

11: $\mathcal{CA} = \text{ComputeAffineSet}(r_i, \mathcal{RC})$

12: cur_min = -1;

13: for all $c_k$ in $\mathcal{CA}$ do

14: $Rc_k \leftarrow r_i$; $\text{SaveandUpdate}(Tc_k)$ // try allocation

15: min_slack = $\text{ComputeMinSlack}(\mathcal{TC})$

16: $Rc_k = Rc_k - r_i$; $\text{Restore}(Tc_k)$ // 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: $\mathcal{G} = \mathcal{G} - r_i$; $\text{Update}(Tc_{tgt})$

22: else

23: if state = firstrun then

24: state = recovery // try recovery once

25: $\mathcal{AR} = \text{ComputeAffineRunnables}(r_i)$

26: $\text{Deallocate}(\mathcal{AR}, \mathcal{RC}); \text{Update}(\mathcal{TC})$

27: $\mathcal{G} = \mathcal{G} \cup \mathcal{AR}$

28: else

29: return fail

30: return success
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 $U_{tot} = \sum C_j U_j$ 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.

Once the list of candidate cores for allocation is selected, the runnable is tentatively allocated to each of them in turn (line 13 to 18). 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 slack (line 20 and 21). Otherwise, the algorithm enters its recovery stage (line 23 to 29). 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 3.3.2.b. 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 8), 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, $A_{\text{cur}}$ 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}($A_{\text{cur}}$) 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).
Algorithm 8 Simulated annealing for task synthesis

1: Construct initial configuration.

2: while $T \leq T^*$ do

3:   while $K < K^*$ do

4:     $A_{new} = \text{randomChange}(A_{cur})$

5:     while existsCycle($A_{new}$) do

6:       $A_{new} = \text{randomChange}(A_{new})$

7:     $C_{new} = \text{memCost}(A_{new})$

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 $P(C_{cur}, C_{new}, T) > \text{rand}()$ then

16:         $A_{cur} = A_{new}$, $C_{cur} = C_{new}$

17:            $K = K + 1$

18:       $T = T \times \beta$

19:    return $A_{opt}$
• 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. `existCycle(A_{new})` will check for the existence of cycles. After each transition, the `memCost(A_{cur})` function computes the memory cost $C_{new}$ of the new candidate solution $A_{new}$, and checks its schedulability using `checkSched(A_{new})`. If the solution is not schedulable, a penalty $\Delta$ 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 $K^* \times$ 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.

### 3.3.3 Experimental Results

As stated in [19], 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 [51], with 58 subsystem blocks, 5 inputs and 10 outputs.
Runnable synthesis: We apply the top-down runnable synthesis method, which completes within two 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 5, we obtain a set of runnable solutions, as shown in Fig. 3.28. Each solution is measured by its modularity metric (number of runnables) and its schedulability metric (alpha ratio).

![Runnable synthesis with top-down method for fuel injection example](image)

Figure 3.28: Runnable synthesis with top-down method for fuel injection example

The leftmost point in Fig. 3.28 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 (3.59)–(3.68). 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 ([43, 44]).

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. 3.28, 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.

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

Table 3.4: Memory costs of the task synthesis results for the fuel injection example (in kB)

Table 3.4 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 one hour for SA and under one minute for GH for each case.

b) Robotics Car Example

The second industrial case study is a Simulink model of a robotics car from [50] 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. 3.29. 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 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 potentially schedulable runnable generations from top-down and bottom-up methods.

Figure 3.29: Runnable synthesis (top-down and bottom-up) for the robotics case
c) Synthetic Examples

We also apply our algorithms to a set of synthetic examples generated by TGFF [24], with 10 to 50 blocks in the diagram (20 examples for each block size).

![Graph showing runnable synthesis with top-down method for synthetic examples]

**Figure 3.30:** Runnable synthesis with top-down method for synthetic examples

**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 one 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. 3.30 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:

![Figure 3.31: Task synthesis algorithms (GH and SA) for synthetic examples](image)

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. 3.31 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 55% – 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% – 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.

### 3.4 Task Period Synthesis - Codesign of Control Systems

As stated in [20], in real-time embedded control systems, the performance of a control function is significantly affected by its sampling period [64, 7] and the end-to-end (sensor-to-actuator) control delay [11, 58, 29]. Smaller sampling periods and end-to-end delays typically provide better control performance, but may jeopardize the system schedulability. Therefore, optimizing period assignments while guaranteeing schedulability is crucial for system performance and reliability. The problem is increasingly important and challenging with the advent of complex control functions and the use of distributed networked platforms. For instance, in automotive electronic systems, the adoption of the *integrated architecture*, where functionality is implemented by multiple tasks and distributed over a network of Electronic Control Units (ECUs) leads to more sharing and contention on computation and communication resources, making the exploration of task and message periods significantly more complex.

The dependency of the control performance on the sampling period can be illustrated intuitively using an example from the Simulink library [2]. Fig. 3.32 (in the next page) shows the model of an automotive electrohydraulic servomechanism controlled by a pulse-width modulated (PWM) solenoid. It is a feedback control loop with sensors sending
Figure 3.32: A Simulink example of a hydraulic servomechanism (representative of a suspension control)

data to a high level controller and then to actuators. The control loop may conceivably be implemented in a distributed fashion, sharing sensors, actuators and computation nodes with other control loops. In the Simulink model, the period of the main control loop is tunable using a workspace variable. Fig. 3.33 shows the comparison of the model performance for sampling periods of 20ms and 100ms, respectively. The graphs show the reference position (in red, dark line) and the actuator position as commanded by the control (in green, light line). At 20ms, the difference between the actual position and the reference position is much smaller, and provides a much better quality of control.

Each control loop, if considered in isolation, typically has a better performance when its sampling period is smaller. However, in resource constrained multiprogrammed systems, reducing the periods of higher priority control tasks may result in larger interference on lower priority control tasks, and lead to longer delays, larger periods and ultimately worse control performance for those lower priority tasks. Finding the optimal configuration with respect to the selection of periods and the assignment of priorities is a very complex
optimization problem that we target in this work.

In this section, we present an algorithm to efficiently explore task and message periods in distributed CAN-based platforms for optimizing control performance while maintaining schedulability. The algorithm enables exploring large design space and facilitates architecture exploration (as more architectural options may be explored with this efficient algorithm). For comparison, we also develop a simulated annealing algorithm and two extensions of the algorithm in [17] (with one directly optimizing latency and the other directly optimizing period). Experiments conducted on an industrial automotive case study and synthetic examples with two different control performance formulations show the advantages of our control-driven approach.
3.4.1 Problem Formulation

In this section, we reuse the formulation proposed in our work in [20] and present here in more details.

a) System model

Our work addresses complex embedded systems where multiple control functions are deployed over a distributed platform consisting of embedded processors connected through communication buses. The control functions run concurrently on the platform and may share computation and communication resources. Architectures of this type are common in the automotive, aeronautics and industrial automation domains.

For instance, Fig. 3.34 shows the functional model of a subsystem that integrates several active safety functions in an experimental vehicle (the same system is used in our experiments). Each active-safety function is implemented by a set of software tasks that communicate through signals. The tasks collect information from various types of sensors (cameras, radars, ultrasound, lidar) on the surrounding objects, the road condition, and the physical environment; apply sensor fusion, filtering and object detection algorithms on the sensor information; decide control actions and less-critical driver assistance functions such as parking assistance; conduct post-processing and arbitration; and eventually send the commands to a set of actuators (brakes, throttle, steer, suspensions, dashboard). Functional dependencies among tasks define a complex graph rather than a set of linear transactions. The system is inherently multi-rate, because of technology constraints (e.g. off-the-shelf sensors operate at different rates) and the usage of multiple control functions governed by
different control laws. Merging control flows with different rates and communication with
oversampling/undersampling is commonplace.

Figure 3.34: Functional model of an experimental vehicle

The distributed architecture platform consists of ECUs (electronic control units)
connected through CAN buses. The ECUs are assumed to run AUTOSAR/OSEK compliant
operating systems that support preemptive priority-based task scheduling, and the standard
CAN bus model features non-preemptive priority-based message scheduling.

118
During implementation, the software tasks in the functional model are mapped onto the ECUs, and the signals exchanged among them are grouped into messages and transmitted on buses. Priorities are assigned to tasks and messages for scheduling. Timing constraints and performance metrics may be defined on local computations and on end-to-end functional paths. In this work, given an allocation and priority assignment for tasks and messages, our approach automatically assign them to optimize the control performance, while satisfying end-to-end deadlines and utilization constraints.

We use a similar system model and notations as in [17]. The mapped system is represented as a directed graph $G = (O, L)$ and a set $R$. $O$ is the set of vertices in $G$ and represents all the tasks (denoted by set $T$) and messages (denoted by set $M$) in the system, i.e., $O = T \cup M$. $L$ is the set of links in $G$ and represents the data dependencies. $R$ is the set of computation resources (ECUs) and communication resources (CAN buses) in the system.

Each object $o_i \in O$ is either a task or a message, and is associated with an activation period $T_i$, a worst case execution/transmission time $c_i$, a priority $p_i$, and a resource $R_{o_i}$ to which it is allocated (an ECU/bus if $o_i$ is a task/message respectively). All objects are scheduled based on their priority (preemptive for tasks and non-preemptive for messages), and a total order exists among the priorities of all objects that are on the same resource. Each $o_i$ has a worst case response time $r_i$, from the activation of the object to its completion (execution for a task or transmission for a message). The computation of $r_i$ needs to take into account the object’s own time requirement $c_i$ and the time spent waiting for the resource access, as explained in Section 3.4.1.c.
Each link $e_i \in \mathcal{L}$ connects a source object and a destination object. One object can be the source or destination for multiple links. When an object completes its execution or transmission, it delivers the data to all outgoing links. For any link, the destination object is activated periodically and reads the latest data transmitted over the link. For any object $o_i$, $\text{Succ}(o_i)$ denotes the set of its successor objects, which are destination objects of $o_i$’s outgoing links. Links do not represent an order of execution constraint, but we assume a communication-by-sampling mechanism in which each task reads the latest data produced by the (cyclic) predecessor task (or delivered by the periodic message).

A path $p$ is a finite sequence of objects that starts from a source object $\text{src}(p)$ and ends with a sink object $\text{snk}(p)$, with a link between every pair of adjacent objects. Sources are activated by external events and sinks activate actuators. Multiple paths may exist between each source-sink pair. The worst case end-to-end latency from $\text{src}(p)$ to $\text{snk}(p)$ is denoted as $l_p$. Deadline requirements may be imposed on selected paths, and the deadline for path $p$ is denoted as $d_p$.

b) Control performance

We assume there are $n$ control functions in the system and the performance of the $i$-th control function is measured by a control cost $J_i$. The objective of our period optimization problem is to minimize the overall system control cost:

$$\min \sum_i w_i J_i \quad (3.73)$$

where $w_i$ is a pre-assigned weight for the $i$-th function.

Each control function may have its own definition of a performance index, which
leads to different formulations of $J_i$. For instance, the performance of a control system for a given time interval $[0, t_f]$ may be defined as in [64]:

$$J = S(x(t_f), t_f) + \int_0^{t_f} L(x(t), u(t), t) dt$$

(3.74)

where $x(t)$ is the system state and $u(t)$ is the control input. $S(\cdot)$ and $L(\cdot)$ are weighting functions that express how the control performance depends on the system states and the control inputs.

In this work, we consider the following two scenarios for a discrete controller. First, we provide a quadratic cost model which takes into account the impact of both sampling period and control delay on control performance. In the second scenario, we consider an exponential decay model to capture the relation of control performance to sampling period only. These two models are independent models under different assumptions. In this paper, we mainly address the quadratic cost model, however the formulations and algorithms can be easily adapted to exponential decay cost model as shown in later sections. Furthermore, our approach can be applied more generally to any non-linear function for which a linear piecewise approximation with bounded error can be found.

**Systems with negligible jitter and single-rate chains** In these systems, control functions are implemented by a set of tasks on a functional path $p$. The source task $src(p)$ of $p$ is a sampling task with period $t_{src(p)}$, which equals to the period $T$ of the control function. Also, the control actuation is applied at the end of each period, with zero jitter. These systems allow a formal analytical treatment for the case in which the control cost is expressed as a quadratic function. This model is common to several papers dealing with the performance evaluation of linear control systems ([30], [29]).
A linear control system can be modeled as:

\[
\dot{x}(t) = Ax(t) + Bu(t) + w(t) \tag{3.75}
\]

where \(w(t)\) is the disturbance, modeled as a zero-mean white Gaussian noise process with covariance function \(E[w(t_1)w(t_2)^T] = Q_c \delta(t_1 - t_2)\).

The state of the system is estimated by a Kalman filter, which makes use of a measurement given by:

\[
x(t) = Hx(t) + n(t) \tag{3.76}
\]

where \(n(t)\) is the measurement error modeled as white, zero-mean, Gaussian noise, with covariance matrix \(R\). We here assume that the above measurement renders the system observable, which ensures that the errors in the state estimates remain bounded.

A control with a zero-order-hold within each period is employed. The discrete-time model for the system can therefore be written as:

\[
x((k+1)T) = \Phi(T)x(kT) + \Gamma(T)u(kT) + w_d(kT) \tag{3.77}
\]

where \(T\) is the sampling period (under our assumption that each control function is single-rate),

\[
\Phi(T) = e^{TA} \tag{3.78}
\]

\[
\Gamma(T) = \int_0^T e^{(T-\tau)A}Bd\tau \tag{3.79}
\]

and \(w_d\) is an error term with covariance matrix

\[
Q_d(T) = \int_0^T e^{T\tau}Q_c e^{\tau A^T}d\tau \tag{3.80}
\]
The objective is to find the control input $u$ that minimizes the cost function:

$$J(NT) = \frac{1}{NT} E \left[ \int_0^{NT} x(\tau) W_1 c x(\tau)^T + u(\tau) W_2 c u(\tau)^T d\tau + x(NT) W_3 c x(NT)^T \right]$$  \hspace{1cm} (3.81)

where $NT$ is the control horizon, and $W_1 c$, $W_2 c$, $W_3 c$ are positive definite weight matrices.

The optimal linear causal controller for the above problem can be computed by a direct application of LQG theory [7]. The control input at each time instant $kT$ is given by:

$$u(kT) = -K_N \hat{x}(kT)$$  \hspace{1cm} (3.82)

where $K_N$ is the control gain, and $\hat{x}(kT)$ is the optimal estimate of the state at time $kT$, computed by a Kalman filter. Since in our system a delay of $m$ periods exists from the time the control input is computed to the time it is applied, the best available estimate of the state at time $kT$ is $\hat{x}_{k|k-m}$, i.e., the estimate of the state at the $k$-th time step, given all measurements up to timestep $k - m$ [47].

Following standard practice, the computation of the control cost is facilitated by considering the infinite horizon cost function:

$$J_\infty(T) = \lim_{N \to \infty} J(NT)$$  \hspace{1cm} (3.83)

which expresses the "expected control cost per unit of time". This is minimized by choosing the control input $u$ as [7].

$$u(kT) = -K_\infty \hat{x}_{k|k-m}$$  \hspace{1cm} (3.84)

with

$$K_\infty = (W_{22} + \Gamma(T)^T S_\infty \Gamma(T))^{-1} (\Gamma(T)^T S_\infty \Phi(T) + W_{12}^T)$$  \hspace{1cm} (3.85)
where $S_\infty$ is the steady-state solution of the control Riccati recursion:

$$S_\infty = \Phi(T)^T S_\infty \Phi(T) + W_{11} - (\Gamma(T)^T S_\infty \Phi(T) + W_{12}^T)^T (W_{22} + \Gamma(T)^T S_\infty \Gamma(T))^{-1} (\Gamma(T)^T S_\infty \Phi(T) + W_{12}^T)$$

(3.86)

and the matrices $W_{11}, W_{12}, W_{22}$ are given by:

$$W_{11} = \int_0^T e^{\tau A} W_{1c} e^{\tau A} d\tau$$

(3.87)

$$W_{12} = \int_0^T e^{\tau A} W_{1c} \Gamma(\tau) d\tau$$

(3.88)

$$W_{22} = \int_0^T (\Gamma(\tau)^T W_{1c} \Gamma(\tau) + W_{2c}) d\tau$$

(3.89)

The optimal value of $J_\infty(T)$ is given by:

$$J_\infty^*(T) = \frac{1}{T} (\text{trace}(S_\infty Q_d(T))) + \text{trace}(K_\infty P_\infty (\Gamma(T)^T S_\infty \Gamma(T) + W_{22}))$$

(3.90)

where $P_\infty$ is the asymptotic value of the covariance matrix of the error in the estimates $\hat{x}_{k|k-m}$ (note that if the system is observable, the state-estimation error covariance matrix converges to a constant value). The matrix $P_\infty$ is given by:

$$P_\infty = \Phi(mT) \bar{P}_\infty \Phi(mT)^T + Q_d(mT)$$

(3.91)

where $\bar{P}_\infty$ is the steady-state solution of the posterior covariance matrix for the state estimate $\hat{x}_{k-m|k-m}$, which is computed by solving:

$$\bar{P}_\infty = ((\Phi(T) \bar{P}_\infty \Phi(T)^T + Q_d(T))^{-1} + H^T R^{-1} H)^{-1}$$

(3.92)

Equation (3.90) provides the control cost as a function of the sampling period $T$ (and sample delay $m$), and is used in (3.73) in order to compute the cost for each of the LQG control functions in our system.
The worst case end-to-end latency $l_p$ of the path is the upper bound of the control delay $l$. According to our assumptions, at runtime we require that the control function waits until $l_p$ to apply the actuation even if the real-time delay is shorter, i.e., we assume $l = l_p$ and sample delay $m = \lceil l_p/T \rceil$. This assumption ensures that the release jitter is always negligible (as in [34]) and can be omitted in the computation of the control performance index.

![Figure 3.35: Quadratic Control Cost Simulation](image)

The simulation result of the quadratic cost is shown in Fig.3.35. The $T$ axis denotes period and $m$ axis denotes delay. It can be seen that larger period and larger delay result in worse control cost (i.e. larger cost value). Because in this work, we assume that the control is applied only on the integer multiple of sampling period, only the integer value is meaningful, in this case, $m$ axis need to be segmented by integer values. The quadratic cost
with integer value of delays ranging from 1-15 is shown in Fig. 3.36. It can be seen again larger period result in larger cost value, the curve on the top corresponds to the largest sample delay (i.e. 15).

![Figure 3.36: Quadratic Control Cost Simulation with Delay Segmented Because of Zero-Order Hold Assumption](image)

Dominant pole and other approximate formulations  For control systems with a dominant pole, the relation between the control performance (measured by a performance loss index) and the control period can be captured by an exponential decay function ([64, 65, 42]):

$$J(T) = \alpha e^{-\frac{\eta}{\tau}}$$  \hspace{1cm} (3.93)
where $\alpha$ is the magnitude coefficient and $\beta$ is the decay rate (both are positive parameters).

Intuitively, the control performance improves ($J$ decreases) when the period $T$ of the control loop decreases.

Figure 3.37: Fitting the RMS of the control error in the Simulink electrohydraulic servo example for different sampling frequencies with an exponential decay function

For the Simulink electrohydraulic servo example shown in the introduction (Fig. 3.32), we measure the control performance as the root mean square (RMS) of the difference between the actuator position and the reference position (i.e., the error in the actuator position). Fig. 3.37 shows this control performance measurement for different sampling frequencies. When the frequency increases (period decreases), the performance gets better with smaller error in the actuator position. An exponential decay function can be used to approximate (fit) the functional dependency of the control performance (RMS of the control error).
error) on the sampling period. As shown in the figure, even when an exponential control cost function cannot be determined analytically, it is still possible to determine the parameters by fitting the cost values obtained through simulation runs for different sampling period values.

More generally, the performance function does not need to be exponential, but could be any non-linear function for which a linear piecewise approximation with bounded error can be found, as explained in section 3.4.2.b. Similar to the previous case, we assume that also for these systems the actuation signals are held until the end of the period.

c) Sampling period and control delay

As defined in Section 3.4.1.b, the sample delay \( m \) depends on the end-to-end latency and sampling period (i.e., \( m = \lfloor l_p / T \rfloor \)). The worst case end-to-end latency \( l_p \) depends on the periods of the objects (including tasks and messages) along the path. An example of the worst case end-to-end latency is shown in Fig. 3.38. In this case, \( \tau_1 \) is a sampling task from \( ECU_1 \), and it is connected with \( \tau_2 \) from \( ECU_2 \) through message \( m_1 \) (allocated on the CAN bus). It can be seen that in the worst case, the latency of the sub-path \( \tau_1 \rightarrow m_1 \rightarrow \tau_2 \) can be much larger than the period of \( \tau_1 \), and therefore the end-to-end path latency can be multiples of \( T_{\tau_1} \).

In the periodic activation model \( l_p \) is computed as:

\[
l_p = \sum_{k: o_k \in P} (T_k + r_k)
\]  

(3.94)

where \( T_k \) is the object period and \( r_k \) is its worst case response time. \( T_k \) is included because of the asynchronous communication between objects: For any link on the path, when the
source object completes and produces output data, the destination object might have just been activated and missed the data. In that case, it will need to wait for the next activation for a time arbitrarily close to its period. For the source task, the arrival of the external event has the similar effect, i.e., it may just have missed the activation of the source task.

The worst case response time of a task is computed as:

\[
r_i = c_i + \sum_{j \in hp(i)} \left\lceil \frac{r_i}{T_j} \right\rceil c_j
\]  

(3.95)

where \( hp(i) \) is the set of higher priority tasks on the same ECU as task \( i \) [36].
The worst case response time of a message can be similarly approximated with the upper bound in [18]:

\[ r_i = c_i + q_i \quad (3.96) \]

\[ q_i = b_i + \sum_{j \in hp(i)} \left\lfloor \frac{q_i + t_{bit}}{T_j} \right\rfloor c_j \quad (3.97) \]

where the blocking time \( b_i \) is upper bounded by the time it takes to transmit the longest (non-interruptible) frame in the system and \( t_{bit} \) is the constant time that is required to send one bit.

Based on (3.95) and (3.96, 3.97), decreasing the period of a task (message) will increase the response time of lower priority tasks (messages) on the same resource, which may make the system unschedulable or increase the end-to-end latencies of other functional paths according to (3.94), and affect the performance of other control functions. Therefore, for a complex system with multiple control functions, it is important to have an integrated formulation that optimizes the control performance while ensuring the schedulability.

d) Schedulability

A feasible period assignment needs to satisfy the resource utilization constraints and schedulability constraints. The utilization constraints set an upper bound on the fraction of time a resource may spend processing its objects. Utilization must not be greater than 100%, and may be further constrained because of the need to ensure future extensibility:

\[ \sum_{i:R_{u_i}=R_j} \frac{c_i}{T_i} \leq u_j \quad \forall R_j \in \mathcal{R} \quad (3.98) \]
where $u_j$ is the utilization bound for the resource.

We assume every task or message needs to complete its computation or transmission within its period (i.e., before the next activation instance of this task or message):

$$r_i \leq T_i \quad \text{(3.99)}$$

and end-to-end latency deadlines apply to selected paths:

$$l_p \leq d_p \quad \text{(3.100)}$$

Finding a feasible period assignment under the above constraints can be formulated as an optimization problem.

### 3.4.2 Control-driven Period Optimization

Our period optimization algorithm leverages GP, which is a special form of convex programming [12] and has polynomial time complexity. A GP in standard form is:

$$\begin{align*}
\text{min.} & \quad f_0(x) \\
\text{subject to} & \quad f_i(x) \leq 1 \quad i = 1, \ldots, m \\
& \quad g_i(x) = 1 \quad i = 1, \ldots, p
\end{align*} \quad \text{(3.101)}$$

where $x = (x_1, \ldots, x_n)$ is a vector of positive real-valued decision variables. $f$ is a set of posynomial functions, and $g$ is a set of monomial functions. A posynomial is the sum of monomials defined as:

$$m(x) = cx_1^{a_1}x_2^{a_2}\cdots x_n^{a_n} \quad c > 0, a_i \in \mathbb{R} \quad \text{(3.102)}$$
The utilization and schedulability constraints can be formulated as posynomials. If the original control cost function in (3.73) could also be formulated as a posynomial, we would be able to formulate the period optimization problem as a GP problem (MIGP to be exact) and solve it using an iterative method similarly as in [17]. However, in general, the control cost function is not necessarily a posynomial. In the remainder of this section, we will introduce our approach to approximate the cost function with a series of posynomials through piecewise linear approximations and formulate a series of MIGP formulations to search the design space.

a) Algorithm flow

Fig. 3.39 shows the flow of our control-driven period optimization algorithm. The original algorithms were presented in our work in [20] and is presented here for detailed explanation. The definition of variables and parameters of the algorithms flow are listed in Table 3.5. In this section, we mainly address the quadratic model, in which the control cost $J_i$ of each control loop is a function of its sampling period $T_i$ and sample delay $m$.

Our proposed algorithms can be easily adapted to the exponential decay model, where the

<table>
<thead>
<tr>
<th>$J_i$</th>
<th>cost of the $i$-th control path</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J_{i,m}$</td>
<td>cost of the $i$-th control path for selected sample delay $m$</td>
</tr>
<tr>
<td>$J'_{i,m}$</td>
<td>piecewise linear function approximated from $J_{i,m}$</td>
</tr>
<tr>
<td>$J'_{i,m,j}$</td>
<td>the $j$-th linear partition of the $i$-th control path for selected sample delay $m$</td>
</tr>
<tr>
<td>$GP_k$</td>
<td>the $k$-th MIGP formulation</td>
</tr>
<tr>
<td>$J'_{GP_k}$</td>
<td>the set of selected linear partitions from each control path which is used to form $GP_k$</td>
</tr>
</tbody>
</table>
control cost only depends on the sampling period.

In our model, the actuation signals are held until the end of the control periods, that is, the actuation is applied at an integer multiple of sampling period, and only integer values of the sampling delay \( m \) need to be considered. Therefore, we divide function \( J_i \) into segments that are associated with different integer values of \( m \). We define \( J_{i,m} \) as a function of the sampling period \( T_i \) for a selected integer delay of \( m \).

We then approximate each \( J_{i,m} \) with a piecewise linear function \( J'_{i,m} \) under a given

---

**Figure 3.39: Control-driven period optimization flow**

---

We then approximate each \( J_{i,m} \) with a piecewise linear function \( J'_{i,m} \) under a given
upper bound on the approximation error. Note that $J'_{i,m}$ is also a function of sampling period $T_i$. The $j$-th linear partition in $J'_{i,m}$ is defined as $J'_{i,m,j}$. We then define $J'_i = \{J'_{i,m,j} | \forall m, j\}$ as the set of all linear partitions (including all possible $m$ and $j$ within the range being considered) for function $J_i$.

The piecewise linear approximation of the control cost enables transforming a generally nonlinear optimization problem into multiple optimization problems that have linear objective functions with non-negative coefficients. Such objective functions are posynomials and the corresponding optimization problems can be casted into MIGP formulations and solved in iterations, as shown in Fig. 3.39. Specifically, in the $k$-th iteration, we select one linear partition $J'_{i,m,j}$ for each $J'_i$ to form an MIGP formulation $GP_k$ ($J'_{i,m,j}$ may be better denoted as $J'_{i,m,k,j,k}$, since $m$ and $k$ together represents one specific linear partition that depends on $i$ and $k$. We drop the subscripts here for simplicity).

In solving the MIGP formulation $GP_k$, we minimize the approximated system control cost $\sum w_i J'_{i,m,j}$ (where $m$ and $j$ are given by the selection of linear partitions, and the set of $J'_{i,m,j}$ being considered is denoted as $J'_{GP_k}$ as shown in Fig. 3.39). We explore $T_i$ and other task and message periods, while constraining $T_i$ to $T_{i,m,j} \leq T_i < T_{i,m,j+1}$ (as defined in the selected partition) and the latency $l_i$ to the interval $mT_i < l_i \leq (m + 1)T_i$ (since the delay is $m$).

Each MIGP formulation $GP_k$ is solved efficiently by using a similar approach as in [17]. Then, after solving multiple MIGP formulations with different selections of linear partitions for each $J'_i$, we may choose the period assignment that provides the best overall control performance while satisfying the design constraints.
In the case of multirate systems with an approximate function, the intervals that are used for the piecewise approximation could be selected without requiring that the latency is an integer multiple of a single period, however, for convenience, we adopt the same algorithm for partitioning the range of the possible intervals and find linear approximations.

Next, we will introduce the details of the algorithm flow, including how to approximate the control cost functions, how to select the linear partitions in a branch-and-bound fashion to improve algorithm efficiency, and how to solve the formulated GP problem.

b) Control cost approximation

A control cost \( J_i \) is in general a nonlinear function of \( T_i \) and \( m \). However, with a delay of \( m \) and within a short interval \( T_{i,m,j} \leq T_i \leq T_{i,m,j+1} \), \( J_i \) may be approximated by a linear function of \( T_i \) under a given error bound:

\[
J'_{i,m,j} = \alpha_{i,m,j} T_i + \gamma_{i,m,j}
\]

(3.103)

where \( j \) simply denotes the \( j \)-th linear partition (note that \( j \) is also the index of period interval). We assume the control cost is non-negative and monotonic with respect to the sampling period (an assumption that holds in most cases). This implies that \( \alpha_{i,m,j} \) is a non-negative coefficient, and \( J'_{i,m,j} \) is a posynomial.

When approximating \( J_i \) with a linear function \( J'_{i,m,j} \) in a local range of values of \( T_i \) for a selected integer \( m \), we make sure the approximation error is within a bound \( \epsilon_{max} \).

For the \( j \)-th linear partition with delay \( m \), the approximation \( e_{i,m,j} \) can be measured using a least square metric:

\[
e_{i,m,j} = \frac{\int_{T_{i,m,j}}^{T_{i,m,j+1}} (J_i - \alpha_{i,m,j} T_i - \gamma_{i,m,j})^2 dT}{(T_{i,m,j+1} - T_{i,m,j})}
\]

(3.104)
Given a cost function $J_i$, an error bound $\epsilon_{\text{max}}$, and lower bound $T_{\text{min}}$ and upper bound $T_{\text{max}}$ for the sampling period $T_i$, Algorithm 9 generates the linear partitions for each segment. The set $P_{i,m}$ contains the values $T_{i,m,j}$ that divide $[T_{\text{min}}, T_{\text{max}}]$ into period intervals for each value of $m$. In other words, given $m$, each period interval $[T_{i,m,j}, T_{i,m,j+1}]$ is a linear partition in which $J_i$ is approximated by a linear function $J'_{i,m,j}$.

**Algorithm 9 linear_approximation($J_i, \epsilon_{\text{max}}, T_{\text{min}}, T_{\text{max}}$)**

1: for all $m \in [1, 2, ..., m_{\text{max}}]$ do
2:   $T_{i,m,0} = T_{\text{min}}$, $j = 0$, $P_{i,m} = \{T_0\}$ \{$P_{i,m}$ stores the values that divide intervals within $[T_{\text{min}}, T_{\text{max}}]$\}
3:   while true do
4:     if $T_{i,m,j} = T_{\text{max}}$ then
5:       $P_{i,m} \cup \{T_{\text{max}}\}$
6:       return
7:     \quad $t_l = T_{i,m,j}$, $t_u = T_{\text{max}}$, $t = T_{\text{max}}$
8:     \quad while $t_u - t_l > \delta$ do
9: \quad \quad $J'_{i,m,j} = \text{linear_regression}(J_i, T_{i,m,j}, t, m)$
10: \quad \quad $e_{i,m,j} = \text{calculate_approximation_error}(J_i, J'_{i,m,j})$
11: \quad \quad if $e_{i,m,j} > \epsilon_{\text{max}}$ then $t_u = t$ else $t_l = t$
12: \quad \quad $t = (t_l + t_u)/2$
13: \quad \quad $j = j + 1$, $T_{i,m,j} = t_l$, $P_{i,m} \cup \{T_{i,m,j}\}$

For each chosen delay $m$, to divide $[T_{\text{min}}, T_{\text{max}}]$ into period intervals, we start with $T_{i,m,0} = T_{\text{min}}$ and use a binary search to find the largest $T_{i,m,1} \in [T_{i,m,0}, T_{\text{max}}]$ that guarantees an approximation error under $\epsilon_{\text{max}}$. Once $T_{i,m,1}$ is selected, we search for the largest $T_{i,m,2} \in [T_{i,m,1}, T_{\text{max}}]$ that satisfies the error bound. The procedure is repeated until all period intervals are defined (i.e., $T_{i,m,j} = T_{\text{max}}$). The details of finding each $T_{i,m,j}$ are shown in Algorithm 9, line 7 to 12. Specifically, $t_l$ and $t_u$ are the lower and upper bound of the next $T_{i,m,j}$ (i.e., $T_{i,m,j+1}$) that is being searched, and $t$ is the value that is evaluated for
$T_{i,m,j+1}$. First, for periods within $[T_{i,m,j}, t]$, we approximate $J_i$ with a linear function $J'_{i,m,j}$ using a linear regression, and compute the approximation error according to (3.104) (line 8 to 10). The approximation error is compared with $\epsilon_{max}$ to update $t_u$, $t_l$ and $t$ for the next round of the binary search (line 11 and 12) until $T_{i,m,j+1}$ is found. Once all linear partitions (i.e., all period intervals for certain delay) and their corresponding linear approximation functions $J'_{i,m,j}$ are found, the piecewise linear function $J'_{i,m}$ for delay $m$ is defined.

Note that in the case of the exponential decay function, there is no delay segmentation step and the outer for loop in Algorithm 9 can be eliminated.

c) Exploration of linear partitions

After every cost function $J_i$ is approximated by a piecewise linear function $J'_{i,m}$ for each segment of delay $m$, we can select one linear partition $J'_{i,m,j}$ for each $J_i$ and approximate the original objective function (3.73) with a linear formulation:

$$\min \sum_i w_i J_i \approx w_i J'_{i,m,j} = \sum_i w_i (\alpha_{i,m,j} T_i + \gamma_{i,m,j})$$ (3.105)

where $T_i$ is the sampling period for $J_i$, constrained within a period interval (i.e., $T_{i,m,j} \leq T_i \leq T_{i,m,j+1}$) with delay $m$ (i.e., $mT_i < l_i \leq (m+1)T_i$, where $l_i$ is the path latency).

We can optimize the period assignments by recursively exploring the selections of linear partitions and solving MIGP formulations, which correspond to the steps after all $J'_{i,m,j}$ are generated in Fig. 3.39.

To obtain the global optimal solution, all possible selections of the linear partitions for each $J'_{i} = \{J'_{i,m,j} \mid \forall m, j\}$ need to be considered. However, simply enumerating all possible selections could be very time consuming for complex systems. Therefore, we
designed a branch-and-bound method to prune the search space during the exploration, as shown in Algorithm 10 and 11.

In Algorithm 10, different control cost functions \( \{J_i \mid \forall i\} \) are first ordered based on the weight \( w_i \) in (3.105) and how sensitive the control cost is with respect to period and delay changes calculated as:

\[
S = w_i \frac{J_i(T_{\text{max}}, m_{\text{max}}) - J_i(T_{\text{min}}, m_{\text{min}})}{(T_{\text{max}} - T_{\text{min}})(m_{\text{max}} - m_{\text{min}})}
\]  

(3.106)

where \( J_i(T_{\text{max}}, m_{\text{max}}) \) (or \( J_i(T_{\text{min}}, m_{\text{min}}) \)) is the control cost \( J_i \) obtained under the maximum (or minimum) period and latency being considered. Then, the control cost functions are re-labeled as \( J_1 \) to \( J_n \) based on this order, which roughly correlates with how much impact a function has on the overall objective, e.g., \( J_1 \) has the most impact.

Starting from \( J_1 \), we recursively explore the selections of linear partitions for each \( J_i \) – more specifically, from each corresponding set of linear partitions \( J_i' \). We define \( J_{GP}' \) as the set of currently selected linear partitions, which later forms the GP problem to be solved.

**Algorithm 10** period_optimization(\( \{J_i \mid \forall i\} \))

1: order_control_functions(\( \{J_i \mid \forall i\} \))

2: \( O_{\text{best}} = \text{MAX} \)

3: \( J_{GP}' = \emptyset \)

4: linear_partition_exploration(\( J_1 \))

Algorithm 11 shows one step of the recursive algorithm for \( J_i \). We explore all possible linear partitions from its corresponding \( J_i' \). Once a linear partition is selected from \( J_i' \), we estimate the lower bound of the control cost in Equation (3.73), based on the current
Algorithm 11 linear_partition_exploration($J_i$)

1: while exist_unexplored_partition($J'_i$) do
2: \( J_{i,m,j}' = \) select_linear_partition($J'_i$)
3: \( J'_GP = J'_GP \cup \{ J_{i,m,j}' \} \) {add selected $J_{i,m,j}'$ to the current GP formulation}
4: \( O_e = \) estimate_objective_lower_bound($J'_GP$)
5: if \( O_e < O_{best} \) then
6: \hspace{1em} if $i = n$ then
7: \hspace{2em} \{ $T_1, T_2, \ldots, T_n$ \} = solve_GP($J'_GP$)
8: \hspace{2em} \( O_c = \) compute_cost(\{ $T_1, T_2, \ldots, T_n$ \}) \{compute cost using original control cost formulation $J_i$\}
9: \hspace{2em} if \( O_c < O_{best} \) then
10: \hspace{3em} \( O_{best} = O_c \)
11: \hspace{3em} record_current_best_solution(\{ $T_1, T_2, \ldots, T_n$ \})
12: \hspace{1em} else
13: \hspace{2em} linear_partition_exploration($J_{i+1}$)
14: \( J'_GP = J'_GP \setminus \{ J_{i,m,j}' \} \)

selection of linear partitions $J'_GP$. If the lower bound $O_e$ is larger than the current best solution $O_{best}$, we stop and explore other linear partitions for $J'_i$; otherwise we continue the exploration. Once one linear partition is selected for each control function, we can form the MIGP formulation and solve it with an efficient algorithm (which will be explained below in Section 3.4.2.d). A good solution may greatly reduce the search space, as observed in our experiments. Please note that $O_e$, $O_c$ and $O_{best}$ are estimated/calculated based on the original objective function in (3.73). Therefore, the exploration process shown in Algorithm 10 and 11 is optimal if the MIGP formulation is solved optimally (in current implementation, the MIGP is approximated for efficiency, as shown next).
d) GP for period optimization

Once a linear partition \( J_{i,m,j}' \) is selected for each function \( J_i \) (as shown in Algorithm 10 and 11), the linear approximation of the cost function (3.105) can be combined with other constraints in an MIGP formulation as follows.

\[
\min \sum_{J_{i,m,j}' \in J'_{GP}} w_i (\alpha_{i,m,j} T_i + \gamma_{i,m,j}) \tag{3.107}
\]

\[
\frac{T_{i,m,j}}{T_i} \leq 1 \quad \frac{T_i}{T_{i,m,j+1}} \leq 1 \quad \forall J_{i,m,j}' \in J'_{GP} \tag{3.108}
\]

\[
\frac{m_i T_i}{l_i} \leq 1 \quad \frac{l_i}{(m+1)T_i} \leq 1 \quad \forall J_{i,m,j}' \in J'_{GP} \tag{3.109}
\]

\[
\frac{T_s}{T_p} \leq 1 \quad \forall o_p, o_s \in O(J_t') \land o_s \in Succ(o_p) \tag{3.110}
\]

\[
\frac{c_i + \sum_{j \in hp(i)} z_{ij} c_j}{r_i} \leq 1 \quad \forall o_i \in T \tag{3.111}
\]

\[
\frac{c_i + b_i + \sum_{j \in hp(i)} z_{ij} c_j}{r_i} \leq 1 \quad \forall o_i \in M \tag{3.112}
\]

\[
\frac{r_i}{T_j \times z_{ij}} \leq 1 \quad \forall o_i \in T \land j \in hp(i) \tag{3.113}
\]

\[
\frac{r_i - c_i + t_{bit}}{T_j \times z_{ij}} \leq 1 \quad \forall o_i \in M \land j \in hp(i) \tag{3.114}
\]

\[
\sum_{i:o_i \rightarrow R_j} \frac{c_i}{T_i \times u_j} \leq 1 \tag{3.115}
\]

\[
\frac{r_i}{T_i} \leq 1 \quad \forall o_i \in \mathcal{O} \tag{3.116}
\]

\[
\frac{\ell_p}{d_p} \leq 1 \quad \forall p \in \mathcal{P} \tag{3.117}
\]

\[
\frac{T_{min}}{T_i} \leq 1 \quad \frac{T_i}{T_{max}} \leq 1 \quad \forall o_i \in \mathcal{O} \tag{3.118}
\]

Equation (3.107) is the linear objective function from (3.105), where \( w_i, \alpha_{i,m,j}, \gamma_{i,m,j} \) are constants. \( T_i \) is the sampling period of \( J_i' \), which corresponds to the period of the sampling task on the control path. \( l_i \) is the control delay of \( J_i' \), which corresponds to the
end-to-end path latency. Constraints (3.108) and (3.109) apply to $T_i$ and $l_i$ based on the selected linear partitions, where $m$ and $j$ represent the selected delay and index of period interval.

Constraint (3.110) requires that for any two consecutive objects on the control path associated with $J_i'$, the period of the successor object is not larger than the period of the predecessor object. This is to ensure that there is no message overwriting, as in the assumptions for our control cost modeling in Section 3.4.1. Constraints (3.111) and (3.112) are reformulations of (3.95) and (3.97), with $z_{ij}$ being the integer variables which denote the preemptions from a higher priority object $j$ on a lower priority object $i$. These integer variables are bounded by (3.113) and (3.114). Constraint (3.115) bounds the utilization as defined in (3.98). Constraint (3.116) encodes the schedulability requirement (3.99), while constraint (3.117) encodes the end-to-end deadline requirements (3.100). Finally, (3.118) defines upper and lower bounds for the periods.

Once the MIGP problem is formulated, we apply the iterative approach from [17] to solve it efficiently. First, the GP approximation takes the integer variables $z_{ij}$ and relaxes them to real-valued variables, and adds parameters $0 \leq \rho_{ij} \leq 1$ to $r_i/T_j$ and $(r_i-c_i)/T_j$ for tasks and messages, respectively. Then, an iterative process runs a GP optimization, and adjusts the $\rho_{ij}$ values until the error between the exact integer number of preemptions and the approximated counts falls below the given threshold. At each step during the iteration, the optimality is approximated, but the correctness of the solution is verified against the original (non-linear) constraints. Since in this work, this iterative process is nested inside the exploration of linear partitions (line 7 in Algorithm 11), GP iterations can be cut short
if the error bounds are too high.

Note that in the case of the exponential decay function, sample delay is not considered and can be eliminated from the GP formulation (i.e., eliminating Equation (3.109)).

e) Algorithm complexity

The complexity of our algorithm depends on the control performance approximation error bound $\epsilon_{max}$, the approximation of MIGP with a series of GP iterations, and the system scale (in particular the number of input blocks). First, $\epsilon_{max}$ determines how many linear partitions need to be explored in theory. As demonstrated in our experiments, the number of linear partitions to be explored increases almost exponentially with the decrease of $\epsilon_{max}$ (i.e., lowering error bound), however the majority of such iterations can be pruned through our branch-and-bound method. The complexity of using GP iterations to approximate MIGP depends on how fast the results may converge. Although it is difficult to have a theoretical guarantee, in practice, the results often converge within a few iterations under the given approximation error bound. Finally, the number of input blocks does not have direct impact on the number of GP iterations required but rather affect the execution time of each GP iteration. In theory, as GP could achieve polynomial complexity with problem size, the complexity of our algorithm is polynomial with respect to the number of input blocks as well. In Section 3.4.4, we provide experimental results regarding the computational complexity of the algorithm.
3.4.3 Comparison with Simulated Annealing

As a comparison with our GP-based approach, we also implemented a simulated annealing (SA) algorithm for period optimization. Simulated annealing relies on random permutations for design optimization and shares some similar characteristics with other randomized search methods such as genetic programming and tabu search.

**Algorithm 12 simulated_annealing\( (t_0, t_F, C) \)**

1. \( \text{curSol} = \text{set\_initial\_solution()} \)
2. \( \text{bestCost} = \text{MAX} \)
3. \( nIter = 0, t = t_0 \)
4. while \( nIter < \text{MAXITER} \land t > t_F \) do
5.   \( nSucc = 0, nTry = 0 \)
6.   while \( nTry < \text{MAXTRY} \land nSucc < \text{MAXSUCC} \) do
7.     \( \text{newSol} = \text{random\_change(\text{curSol})} \)
8.     \( \text{newCost} = \text{evaluate\_cost(\text{newSol})} \)
9.     if \( \text{accept\_change(\text{newCost}, \text{curCost}, t)} \) then
10.       \( \text{commit\_change(\text{newSol}), curCost = newCost} \)
11.       \( nSucc ++ \)
12.     if \( \text{curCost < bestCost} \) then
13.       \( \text{bestSol = curSol, bestCost = curCost} \)
14.     \( nTry ++ \)
15.   \( t = t \times C \)
16. \( nIter ++ \)
Algorithm 12 shows our approach based on the standard SA procedure. We start with an initial period assignment that sets the sampling periods on the targeted control paths to $T_{\text{min}}$ and sets the object periods on the other paths to the value of $(\text{path deadline} / \text{number of objects on the path})$. We found this assignment in general provides better results than a purely random assignment. Starting from the initial period assignment and an initial temperature $t_0$, we use a random transition operator iteratively to search the design space (line 4 to 16), until the number of iterations exceeds a limit or the temperature $t$ is lower than the predetermined final temperature $t_F$.

For each transition to a new solution, we randomly select a task and change its period, and then evaluate the cost of the new configuration (line 7 and 8). The cost is calculated based on the value of the original objective function (3.73), and also a penalty on constraint violations (e.g. violations on latency and utilization constraints), since the intermediate period assignments in SA might not satisfy the design constraints. The random move will be accepted if the new cost is not greater than the current cost, or accepted with a probability $P = e^{(\text{curCost} - \text{newCost})/t}$ if it is greater, as evaluated in the function `accept_change()`. The temperature $t$ is relatively high initially to allow jumping out of local minima, and is lowered with a cooling rate $C$ as the search proceeds and eventually becomes a local search.

### 3.4.4 Experimental Results

As stated in [20], we conduct experiments on an industrial automotive case study and a set of synthetic examples to evaluate the performance of our control-driven period optimization algorithm, and compare it with other approaches, including the original GP-
based approach that focuses on schedulability [17], two extensions of the original GP-based approach that we developed, and the simulated annealing solution in Section 3.4.3. We evaluate two types of control cost formulations introduced in Section 3.4.1, including the quadratic control cost and the exponential decay function. The experiments are run on a 2.9GHz dual-core CPU with 8G memory.

a) Industrial automotive case study

First, we choose three automotive application examples from the Simulink library, study their control performance under different sampling periods through simulations, and formulate the control cost functions by fitting the simulation data. In these three examples, the control cost can be closely approximated with an exponential decay function of the sampling period:

- The electrohydraulic servo example shown in the introduction. The initial fitting of the control cost (measured by the RMS of the control error) versus the sampling period is shown before in Fig. 3.37. We normalize the control error and formulate a control cost function in the exponential decay form as in Equation (3.93). The exponential fitting is very close to the simulation data, with an R-squared value of 0.976 (1 is a perfect fit).

- A fuel control system example. The closed-loop controller detects the amount of residual oxygen in the exhaust gas from a sensor, and determines the fuel rate to keep the air-fuel ratio close to the target ratio. We choose the RMS of the difference between the actual air-fuel ratio and the target ratio as the control cost, and measure
it under different sampling periods. The relation between the control cost and the period is fitted with an exponential decay function, with an R-squared value of 0.994.

- An engine speed control example. The speed controller detects the current speed from the sensor, and applies a PID control based on its difference with the reference speed. Through simulations, we observe that the engine speed response becomes oscillatory as the sampling period increases. We use the standard deviation of the engine speed to measure the control performance. The control cost function is fitted with an exponential decay function of the period and the R-squared value is 0.996.

We consider the case where the three applications above are implemented on a distributed automotive platform for an experimental vehicle. The architecture consists of 29 ECUs connected with 4 CAN buses. 92 tasks are executed on the ECU nodes, and 196 messages are exchanged over the buses. The system graph contains a total of 222 functional paths, many of which share common tasks. End-to-end deadlines are placed over paths between 12 pairs of source-sink tasks in the system. The deadline is set at 300ms for nine of these source-sink pairs, at 200ms for two pairs, and at 100ms for one pair. The upper bound and lower bound of the task periods are set to 100ms and 5ms, respectively.

We assume the three automotive control applications above are mapped onto three functional paths with distinct source and sink tasks. For any of the three applications, the period of the corresponding source task determines its control cost, based on the exponential decay function we build from the Simulink simulations. We conduct a set of experiments, in each of which we select a subset of the three applications and optimize the total control cost as defined in the original objective function (3.73), with the weights $w_i$ set to 1 and
the $J_i$ set to the formulated exponential decay functions. Note that in the optimization, the end-to-end deadlines for all 222 paths need to be satisfied.

In each experiment, we compare our control-driven GP-based approach (denoted as “CDGP”) with three other approaches that do not address the control cost directly:

- The original GP-based approach in [17], where the sum of the object response times is used as the optimization objective (denoted as “RGP”).
- A modification of the approach in [17], where the sum of the end-to-end latencies for the target control paths is the optimization objective (denoted as “LGP”).
- Another modification where the sum of the object periods on the target control paths is used as the optimization objective (denoted as “TGP”).

LGP and TGP are simple extensions of RGP that we developed in this work. Although they do not consider the control cost directly, each of them addresses an essential element of the control cost – latency in LGP and period in TGP, respectively.

The results of the experiments are shown in Fig. 3.40 and Table 3.6. Fig. 3.40 shows the normalized control costs for CDGP, LGP and TGP under different scenarios, where the costs for CDGP are assumed as the base: “EH+FC+EC” is an objective function by which all three applications are optimized, i.e., using $J_1+J_2+J_3$ as the objective; “FC+EC” means that we only optimize the fuel control application and the engine control application; and so on. Our CDGP approach provides better solutions than the LGP and TGP, and the relative improvement increases when all three applications are being optimized (in which case the correlation between the total control cost and the period assignment is more complex and
Figure 3.40: CDGP vs. LGP and TGP for industrial case study with exponential control cost functions

the optimization is harder).

Table 3.6 shows the comparison between normalized CDGP and RGP costs on the same set of experiments, with CDGP as the base. The RGP approach from [17] does not address either period or latency in its objective function, and provides solutions that have much higher costs than CDGP (therefore is not plotted in Fig. 3.40 for readability). Overall, the results from Fig. 3.40 and Table 3.6 show the importance of optimizing with an objective function that directly addresses the control cost, rather than using indirect objectives such as latencies, periods, or response times.
Table 3.6: CDGP vs. RGP for industrial case study with exponential control cost function

<table>
<thead>
<tr>
<th>Scenario</th>
<th>EH</th>
<th>FC</th>
<th>EC</th>
<th>EH+FC</th>
<th>EH+EC</th>
<th>FC+EC</th>
<th>EH+FC+EC</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDGP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>RGP</td>
<td>7.1</td>
<td>70.8</td>
<td>22.7</td>
<td>15.2</td>
<td>10.9</td>
<td>37.7</td>
<td>18.5</td>
</tr>
</tbody>
</table>

b) Synthetic examples

We conducted a set of experiments with randomly generated cases from the SMFF (System Models For Free) tool [53]. We generate 50 system configurations, with 5 to 22 ECUs, 12 to 99 tasks, and 6 to 54 messages.

First, we consider the exponential decay functions and conducted experiments with 1 to 5 selected control paths/functions (10 test cases for each number of control functions and the results are averaged). The bounds for the task periods and the parameters of exponential decay functions are similar as in the industrial case study. Fig. 3.41 shows the normalized control cost for CDGP, LGP and TGP. Similarly as in the industrial case study, our CDGP approach provides much smaller control costs than LGP and TGP when the number of control functions being optimized becomes larger. The results from RGP are much worse (9X to 69X of the CDGP costs) and are omitted here.

We also consider the cases for a quadratic cost function. Fig. 3.42 shows the normalized results for CDGP, LGP and TGP. CDGP still provides better solutions than LGP and TGP (the latter are almost identical). However, compared with exponential functions, the improvement is much smaller. The RGP results are still significantly worse, with 2.54X to 268.93X of the CDGP costs.
c) **Comparison with SA**

We also compared our approach with a Simulated Annealing (SA) implementation. SA and other randomized search methods, such as genetic algorithms are commonly used in design space exploration. Comparison with it can provide a valuable assessment of the effectiveness of our approach.

First, we conducted experiments for the industrial test case and selected 1 to 3 control paths for optimization. The CDGP can find a feasible solution within 0.5 hour and eventually finds the optimal solution (within the linear approximations). The SA algorithm
Figure 3.42: CDGP vs. LGP and TGP for synthetic examples with quadratic control cost function

cannot find any feasible solution within 24 hours for any number of control paths (e.g., in the case of 1 control path, 4 task deadline constraints are still violated when SA timed out).

For further comparison, we scaled down the task computation times to ease finding feasible solutions. The results for the new case study are shown in Table 3.7. The CDGP approach provides slightly better results than SA, with much faster runtimes (53X to 300X).

We conducted similar experiments for the synthetic examples with both exponential decay and quadratic cost functions, and observed similar results: CDGP provides
Table 3.7: CDGP vs. SA for industrial case study (task computation time halved)

<table>
<thead>
<tr>
<th>Path No.</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDGP norm. cost</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SA norm. cost</td>
<td>1.07</td>
<td>1.04</td>
<td>1.11</td>
</tr>
<tr>
<td>CDGP runtime (hr)</td>
<td>0.08</td>
<td>0.18</td>
<td>0.45</td>
</tr>
<tr>
<td>SA runtime (hr)</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
</tbody>
</table>

better costs than SA (in average 9.1% smaller) with much faster runtimes (10X to 30X). Overall, we observed that 1) when the system utilization is high and constraints are tight, using CDGP allows to more easily find feasible solutions since it can efficiently search the design space globally while SA may be stuck at local optima for a long time; 2) when the system utilization is low and constraints are easy to meet, using CDGP provides similar optimization results as SA but with a much faster runtime. Furthermore, the closeness of the results from the two algorithms suggests that the obtained solutions may be very close to the true optimum (although no formal proof can be drawn unless all configurations are evaluated, which is intractable for practical cases).

d) Impact of function weights

In the above experiments, the control function weights $w_i$ are set to 1, i.e., all control functions have the same weight. We also conducted experiments with different weights values, as in practice control loops might have different levels of importance. Specifically, we chose two control paths from the industrial test case. The objective function is therefore $w_1 J_1 + w_2 J_2$. We set $w_2$ to 1 and $w_1$ to different values between 1 and 10. The results are in Table 3.8. CDGP provides better results than SA and the difference gets larger when the weights are more different.
### Table 3.8: CDGP and SA with different function weights

<table>
<thead>
<tr>
<th>$w_1$ weight</th>
<th>1</th>
<th>2.5</th>
<th>5</th>
<th>7.5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDGP norm. cost</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SA norm. cost</td>
<td>1.040</td>
<td>1.053</td>
<td>1.062</td>
<td>1.065</td>
<td>1.067</td>
</tr>
</tbody>
</table>

**e) Algorithm runtime and scalability**

For both exponential and quadratic cost functions, CDGP is much more efficient than the simulated annealing algorithm (10X to 300X as explained in Section 3.4.4.c). We believe the efficiency of CDGP can be further improved with tighter interaction between the branch-and-bound exploration and the inner GP solving (currently the pruning during branch-and-bound provides 4X to 157X speedup).

The runtime of our CDGP algorithm is significantly impacted by the number of input blocks and the number of linear partitions being explored for each $J_i$. For an exponential cost function, the number of linear partitions is relatively small since the cost does not depend on the control delay (therefore only period intervals need to be explored). Next, we conduct more experiment to analyze the runtime of our CDGP algorithm for quadratic control cost function.

**$\epsilon_{max}$ vs. runtime** The control performance approximation error bound $\epsilon_{max}$ has a significant impact on the number of linear partitions to be explored. We conduct scalability analysis with respect to the increase of $\epsilon_{max}$. Fig. 3.43 shows the total number of linear partitions to be explored in theory and the actual number of linear partitions being explored after pruning in our branch-and-bound method, with respect to different $\epsilon_{max}$ values (normalized). We can see that the number of linear partitions to be explored in theory...
decreases almost exponentially with the increase of $\epsilon_{max}$ (until it saturates). However, the actual number of linear partitions being explored is a much flatter curve. In other words, when decreasing $\epsilon_{max}$ (i.e., lowering error bound), the number of linear partitions to be explored increases almost exponentially in theory, however the majority of such partitions can be pruned in our branch-and-bound method. In fact, in average 98% of such partitions can be pruned with up to 58X speedup. This demonstrates the effectiveness of our branch-and-bound method.

![Figure 3.43: The number of linear partitions to be explored in theory and after pruning with different selections of $\epsilon_{max}$](image)

**Input size vs. runtime** The runtime of our CDGP algorithm under different input sizes (i.e., the number of input blocks) is shown in Fig. 3.44. We can see that the runtime
increases almost linearly with respect to the input size, which is consistent with our analysis in Section 3.4.2.

![Figure 3.44: CDGP runtime with different selections of input size (measured by the number of blocks)](image)

**Impact of Non-control Tasks and Messages** We further evaluate the effectiveness of our approach when there are non-control (or legacy) tasks and messages with *fixed periods* in the system. In our original synthetic examples, when there are only control tasks with non-fixed periods in the system, the utilizations of ECUs after optimization are between 14% to 52%. We then gradually increase the load of fixed-period tasks until the ECU utilizations reach 34% to 70% (further increase will violate the deadline constraints), and assume they have lower priorities than the non-fixed-period control tasks. We observe that our approach
remains very effective in optimizing the periods of control tasks and improving control performance. The control objective function only decreases by 3.6% in average. There is similar trend when increasing the load of fixed-period messages.

If we randomly assign priorities to fixed and non-fixed tasks and messages, the control performance may decrease significantly when the load of fixed-period tasks and messages increases. This is expected, since in this case the response times of lower-priority control tasks and messages are increased and their periods may have to be large.
Chapter 4

Conclusions and Future Works

4.1 Conclusion

In this dissertation, we presented an integrated flow for synthesizing software implementations from synchronous reactive model for cyber-physical systems. We developed algorithms and methodologies to automate and optimize the generation of software tasks from functional model and the allocation and scheduling of tasks on hardware platforms. To summarize, the major characteristic of the proposed approaches includes:

1) The proposed flow closes the gap between task generation and task mapping, as we integrate them in a holistic process with timing consideration throughout. Section 3.3 is a demonstration of this integrated flow, in which a unified metric (FETA) is proposed for timing analysis in task generation and task mapping, and a complete flow starting from functional model to runnables and then to tasks on hardware platforms are proposed.

2) As timing is critical to system performance as well as functional correctness. We consider it together with other software engineering objectives throughout the synthesis flow.
We developed algorithms for optimizing various timing based metrics including schedulability, extensibility, robustness, and various software engineering objectives including, reusability, modularity, code size and memory, etc. The experimental results demonstrated that our synthesis algorithms are able to provide efficient, correct, extensible, modular and reusable software implementations for cyber-physical systems. This will significantly improve design quality and productivity.

3) We developed a codesign framework for control-centric CPS applications that optimizes control performance while satisfying stringent system timing constraints. Due to the flexibility of the formulation, the framework can be easily modified to incorporate other design metrics.

4) In addition to automotive systems, we believe our methodologies can be applied to other CPS application domains such as avionic systems, industrial automation and robotic systems (with customization). In particular, the proposed methods addresses synchronous reactive model, which is widely used for the modeling of control-centric CPS applications.

4.2 Future Works

There are several directions for future works.

- Current synthesis flow mainly focus on timing and other software engineering metrics. It could be extended to other important design objectives, including fault-tolerance, security, etc.

- We can extend the codesign frameworks to other CPS domains. For instance, vision
algorithms are being increasingly used in CPS domains, e.g. autonomous driving. In resource constrained system, codesign of vision-centric CPS can be quite challenging and important.
Bibliography


