Tool Integration for Automated Synthesis of Distributed Embedded Controllers*

DEBAYAN ROY, Technical University of Munich, Germany
LICONG ZHANG, Technical University of Munich, Germany
WANLI CHANG, Hunan University, China
DIP GOSWAMI, Eindhoven University of Technology, Netherlands
BIRGIT VOGEL-HEUSER, Technical University of Munich, Germany
SAMARJIT CHAKRABORTY, University of North Carolina at Chapel Hill, USA

Controller design and their software implementations are usually done in isolated design spaces using respective COTS design tools. However, this separation of concerns can lead to long debugging and integration phases. This is because assumptions made about the implementation platform during the design phase – e.g., related to timing – might not hold in practice, thereby leading to unacceptable control performance. In order to address this, several *control/architecture co-design* techniques have been proposed in the literature. However, their adoption in practice has been hampered by the lack of design flows using commercial tools. To the best of our knowledge, this is the first paper that implements such a *co-design* method using commercially available design tools in an automotive setting, with the aim of minimally disrupting existing design flows practiced in the industry. The goal of such co-design is to *jointly* determine controller and platform parameters in order to avoid any *design-implementation gap*, thereby minimizing implementation time testing and debugging. Our setting involves distributed implementations of control algorithms on automotive electronic control units (ECUs) communicating via a FlexRay bus. The co-design and the associated toolchain *Co-Flex* jointly determines controller and FlexRay parameters (that impact signal delays) in order to optimize specified design metrics. Co-Flex seamlessly integrates the modeling and analysis of control systems in MATLAB/Simulink with platform modeling and configuration in SIMTOOLS/SIMTARGET that is used for configuring FlexRay bus parameters. It automates the generation of multiple *Pareto-optimal* design options with respect to the quality of control and the resource usage, that an engineer can choose from. In this paper, we outline a step-by-step software development process based on Co-Flex tools for distributed control applications. While our exposition is automotive specific, this design flow can easily be extended to other domains.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; • **Software and its engineering → Software development techniques**; • **Networks → Cyber-physical networks**; • **General and reference → Design**.

Additional Key Words and Phrases: co-design, cyber-physical systems, toolchain, real-time scheduling, embedded control, design automation

*This paper builds on an earlier publication entitled *Multi-Objective Co-Optimization of FlexRay-Based Distributed Control Systems* that appeared at the *2016 IEEE Real-Time and Embedded Technology and Application Symposium (RTAS).*

Authors' addresses: Debayan Roy, debayan.roy@tum.de, Technical University of Munich, Germany; Licong Zhang, licong.zhang@tum.de, Technical University of Munich, Germany; Wanli Chang, wanli.chang.rts@gmail.com, Hunan University, China; Dip Goswami, D.Goswami@tue.nl, Eindhoven University of Technology, Netherlands; Birgit Vogel-Heuser, vogel-heuser@tum.de, Technical University of Munich, Germany; Samarjit Chakraborty, samarjit@cs.unc.edu, University of North Carolina at Chapel Hill, USA.

XXXX-XXXX/2021/1-ART1 $15.00
https://doi.org/10.1145/3477499

ACM Reference Format:
Debayan Roy, Licong Zhang, Wanli Chang, Dip Goswami, Birgit Vogel-Heuser, and Samarjit Chakraborty. 2021. Tool Integration for Automated Synthesis of Distributed Embedded Controllers. *ACM Transactions on Cyber-Physical Systems* 1, 1, Article 1 (January 2021), 30 pages. https://doi.org/10.1145/3477499

1 INTRODUCTION

Software-based implementations of controllers are becoming increasingly more common in domains like avionics, automotive and industrial automation. For example, in the automotive domain, control functions like steering and braking are gradually moving from traditional mechanical or hydraulics systems to electronics and software. These applications are typically implemented on a distributed electrical and electronic (E/E) platform, where a number of electronic control units (ECUs), sensors, and actuators are connected via communication buses such as Ethernet, FlexRay, and CAN. Hence, a distributed embedded control application is partitioned into several software tasks mapped on different ECUs and these tasks communicate via messages sent over the bus. The design of such applications involves two different phases, viz., controller design and platform design. Controller design determines the control law, its parameters, and the appropriate sampling period for an application. Platform design, among other things, computes the task and message schedules.

Conventionally, the platform and the controllers are first designed in isolated design spaces and then integrated [27, 29]. In this approach, the values of the sampling period and the closed-loop delay assumed during the controller design might not be satisfied in the actual platform implementation (i.e., the tasks and/or messages are not schedulable). Similarly, during the platform implementation, it might be assumed that a small change in the periods (or priorities) of the tasks and/or messages will not lead to a significant degradation in the quality of control (QoC). Such assumptions might lead to an error-prone design or long debugging and integration phases. Therefore, to guarantee the safety of the system, engineers often make more conservative assumptions, resulting in less efficient designs. However, as the size and the complexity of systems increase, both computation and communication resources are becoming scarce, making resource-efficient design increasingly important. To address this problem, there has been work [15, 36, 38, 39] on platform and control *co-design.* In contrast to the conventional principle of separation of concerns, co-design approaches try to integrate the design of platform and controllers in an early design phase and exploit the characteristics on both sides to arrive at a more efficient design. Typically, the objectives are to achieve better QoC and minimize resource usage.

Although there is consensus on the advantages of co-design techniques, state of the art co-design methods are far away from the state of practice [47]. The main reason for this is that tools used for controller design and those used for platform design are separate and, more importantly, they are often from different suppliers. Each tool is a product of years of experience in a specific domain. Tool developers and users mostly have a particular set of expertise. Thus, it is challenging to extend one tool and incorporate the functionalities of another from a different domain. An integrated tool flow requires strong collaboration among tool suppliers from different domains [46].

In this context, we consider, as an example, FlexRay-based ECU (electronic control unit) networks from the automotive domain and studied the implementation of controllers on such a platform. We propose a toolchain that enables the development of FlexRay-based systems. This toolchain consists of MATLAB/Simulink for the modeling, design, and analysis of control systems, and SIMTOOLS/SIMTARGET toolboxes [4, 41, 42] for platform modeling and configuration. We first studied the conventional design flow of automotive embedded controllers using such a toolchain. In this work, we then integrated these tools to support a control/platform co-design scheme [36]. In other words, we propose an integrated toolchain to addresses the challenges faced when extending

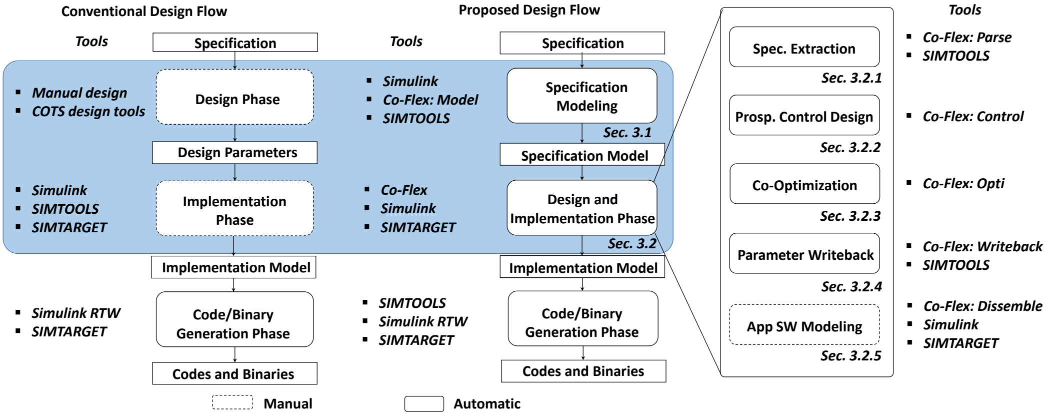


Fig. 1. Schematic of the proposed frame work and the toolchain support

co-design methods to practice in an industrial setting. When compared to a conventional one, our proposed toolchain enables a more convenient and efficient design flow (see Fig. 1 for a comparison).

Conventional design flow: Traditionally, control software development can be divided into three phases: (i) the *design phase* – which involves the calculation and validation of parameters like the control gains, and task and message schedules based on some specification (such as plant models, performance criteria, and an architecture model); (ii) the *implementation phase* – where the system and application software are modeled and configured using the parameters obtained from the design phase; and (iii) the *code generation phase* – where the implemented models are used to generate code and binary files for deployment on a hardware platform. The available toolchains automate certain parts of this development process, e.g., (i) SIMTOOLS/SIMTARGET provide specific blocksets that enable modeling of ECUs and a FlexRay network, partitioning and mapping of tasks, packing of messages into frames, configuration of task and message schedules, and defining input and output interfaces; and (ii) the Simulink Realtime Workshop (RTW) along with SIMTARGET can be used to generate C-code and binary files.

Nevertheless, using such tools, software development also involves the following manual processes and is time-consuming and error-prone. (i) In the design phase, the specification needs to be interpreted to manually formulate the parameter synthesis problem that can then be solved either manually or using some COTS tools. For example, the controller can be designed with a MATLAB/Simulink model of the plant using a closed-loop simulation of the plant and the controller. Here, control gains can be manually tuned, and the ones corresponding to which the closed-loop system meets performance requirements, are chosen as design parameters. On the other hand, the schedule synthesis problem can be formulated manually as a constraint satisfaction problem (CSP) [52] with schedulability constraints, data dependencies, and assumptions made on the values of sampling periods and delays for the controller design. The CSP is then solved using a solver like Gurobi or Z3. In case no feasible schedule configuration exists, the design steps are reiterated. Note that for such an iterative development process, it is very time-consuming to explore the trade-offs between different objectives. (ii) In the implementation phase, the application model is manually developed in Simulink including the details of sensing, control and actuation. The control gains and sampling periods computed in the design phase will be used here. Simultaneously, SIMTOOLS is used to manually specify the platform architecture, how the application tasks are partitioned and mapped on the ECUs, and the schedules of tasks and messages.

Proposed design flow: To address the shortcomings of the conventional flow, in this paper we introduce a toolbox *Co-Flex* based on MATLAB/Simuink and SIMTOOLS/SIMTARGET. Co-Flex will assist software developers by bridging the gap between the aforementioned COTS tools, automating most of the manual steps, while offering more design freedom. Towards reducing manual intervention, Co-Flex offers (i) template blocks that can be conveniently used to model automotive control applications through easy parametrization, and (ii) specific tools that automate the flow between different design phases, e.g., specification extraction from template models, configuration of the control models with the obtained values of control parameters, and synthesizing the implementation model with correct platform parameters (i.e., the task and message schedules). In addition, Co-Flex employs a co-design technique for simultaneously synthesizing the control and the platform parameters by accounting for different trade-offs between QoC and resource usage, thereby offering more design choices.

With Co-Flex, the first two phases in the conventional design flow can be replaced by a *specification modeling* phase and a *design and implementation phase*, as shown in Fig. 1. This is done to reduce the manual effort in the design and implementation phases of a conventional flow. In the specification modeling phase, *Co-Flex: Model* blocksets can be used together with Simulink/SIMTOOLS/SIMTARGET to develop a template software model that is configured according to a design specification, i.e., controlled plant models, architecture model, and performance requirements. Subsequently, the design and implementation phase is composed of five stages, as shown in Fig. 1.

In the first stage, i.e., *Specification Extraction*, the *Co-Flex: Parse* tool can be used to automatically extract the specification from the template model. Stages 2 and 3, called *Prospective Control Design* and *Co-Optimization* respectively, implement the co-design technique in [36]. Such a partition is necessary to reduce the problem complexity by dividing the whole design space into smaller subspaces while considering all feasible regions in the design space. Here, the partitioning is possible because in a FlexRay-based distributed implementation, only a set of predetermined sampling periods are allowed for a control application. Moreover, only the sampling period and not the control gains influences the choice of platform parameters. Thus, in the prospective control design stage, the *Co-Flex: Control* tool is invoked for each application that synthesizes an optimal controller at each possible sampling period. This is done by using a pole placement controller design method and exploring the design space of possible pole values using an exhaustive search with a certain granularity. By designing the prospective controllers first, we avoid unnecessary schedule synthesis for sub-optimal or unstable controllers. In the co-optimization stage, the *Co-Flex: Opti* tool formulates a bi-objective optimization problem according to the extracted specification and the obtained prospective controllers from Stages 1 and 2 respectively. It employs a hybrid optimization technique to generate sets of feasible design parameters, where each set represents a Pareto point reflecting the trade-off between the objectives of QoC and resource usage. Here, the resource usage can only take a finite number of discrete values. Exploiting this fact, the bi-objective optimization problem is transformed into a finite series of single-objective optimization problems, where in each problem, the QoC needs to be optimized for a given resource usage. To solve each problem, a nested two-layer technique is used. This technique exploits the fact that only the choice of sampling periods will influence the QoC and, thus, solves the optimization problem in two nested layers. The outer layer finds the set of sampling periods that optimizes the QoC, whereas the inner layer finds a corresponding set of feasible task and message schedules. In both layers, linear programming problems are solved. It is to be noted here that the co-design uses standard techniques, i.e., the pole placement for controller design and linear programming for time-triggered scheduling, however, the main novelty lies in determining the glue between these techniques.

Subsequently, the developer can select a parameter set corresponding to a Pareto point on the Pareto front according to existing design requirements. Based on the developer's choice, in the *Parameter Writeback* stage, *Co-Flex: Writeback* tool can automatically interpret the synthesis results obtained from the prospective control design and the co-optimization stages respectively and configure the software model with the appropriate values of control and platform parameters. Finally, in the *Application Software Modeling* stage, the *Co-Flex: Dissemble* tool gets rid of the specification models that were required only for the design and need not be a part of the implementation. In addition, the developer can manually add some application-specific details to the model, if required.

Note that the underlying mathematical framework for the co-design of controllers and their platform implementations was originally published in [36]. The main contribution of this paper is a toolchain that implements this co-design framework and integrates it with a combination of industry-strength tools used in real-life design. To the best of our knowledge, Co-Flex is the first published co-design toolchain. We believe that it will motivate the adoption of co-design schemes in the industry, which is crucial for safety-critical and resource-constrained systems.

Contributions: In summary, this paper makes the following contributions:

- We propose a design flow for FlexRay-based distributed control systems that relies on control-platform co-design. In this flow, we start with a specification and first create a partial model of the system. Using this partial model, we synthesize the design parameters that are then used to model the remaining parts of the system. Software code generated from the developed model can be directly used to flash the ECUs.
- We have developed a toolchain to automate the software development for FlexRay-based distributed control systems using the above design flow. This toolchain enables automated modeling of distributed control systems through easy parametrization. It comprises tools implementing control-platform co-design [36], using which a set of Pareto-optimal design options is generated. The toolchain integrates industrial-strength development tools, i.e., MATLAB/Simulink for the modeling and analysis of control systems, with SIMTOOLS/SIMTARGET for the platform modeling and configuration.
- We present a case study comprising five control applications mapped on to three different ECUs communicating over a FlexRay bus. The model-in-the-loop simulation that is offered by SIMTOOLS validates the control and platform parameters synthesized using our co-design scheme. Further, we built a setup comprising three Elektrobit ECUs connected by cables (unshielded twisted pair) and D-SUB9 connectors. We flashed the software binaries on the three ECUs without any errors, implying that the configuration of the design parameters was correct.

Paper organization: In the next section, we explain the feedback control system model and the FlexRay-based ECU network architecture. Further, we describe how feedback controllers are conventionally designed and implemented on such distributed platforms. In Sec. 3, we describe of our design flow along with the proposed toolchain. Next, in Sec. 4, the results based on a case study are presented. Finally, Sec. 5 discusses related work, before concluding in Sec. 6.

2 PRELIMINARIES

We consider a distributed platform comprising a set of ECUs, denoted by $\mathcal{E} = \{E_1, E_2, \cdots\}$. These ECUs are connected by a communication bus. A number of control applications, $C = \{C_1, C_2, \cdots\}$, run on such a *platform*. Each application is implemented using several software tasks performing functions like sensing, computation, and actuation. When these tasks are mapped on physically distributed ECUs, data between them are transferred on the bus. In this work, we study the implementation of feedback control on a FlexRay-based ECU network. In this context, this section

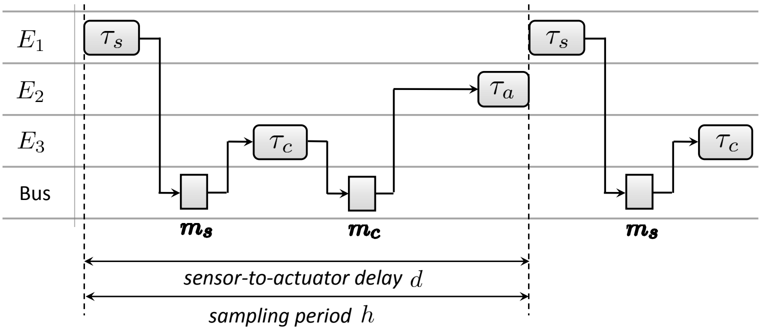


Fig. 2. Distributed embedded control application.

first discusses feedback control systems and the platform architecture under study. Further, we outline the methodology that is commonly followed for developing distributed control software.

2.1 Feedback Control Systems

Plant model: We study linear and time-invariant (LTI) single-input single-output (SISO) systems for which the continuous-time mathematical model can be written as follows:

$$\dot{x}(t) = A^c x(t) + B^c u(t), \qquad y(t) = C^c x(t). \tag{1}$$

Here, $x(t)$, $y(t)$, and $u(t)$ denote respectively the state, the output, and the control input of the system, and A^c, B^c, and C^c denote respectively the state, the input, and the output matrices. For the digital implementation of controllers, we use the zero-order-hold (ZOH) sampling [32, 33]. That is, the system state $x[k]$ is read and the control input $u[k]$ is computed every h time units. Thus, the control input, once computed, is applied to the system for h time units. While there are different techniques available for discretizing a continuous-time signal, ZOH is the most appropriate for studying controller implementations on embedded platforms. Assuming a zero delay between the sampling and the actuation, we can write the sampled-data model of a system as follows:

$$x[k+1] = A^d x[k] + B^d u[k], \qquad y[k] = C^d x[k], \tag{2}$$

where $x[k]$, $y[k]$, and $u[k]$ denote the state, the output, and the control input respectively at the k-th sampling instant ($k \in \mathbb{Z}^*$). The discrete-time system matrices are derived as follows:

$$A^d = e^{A^c h}, \quad B^d = \int_0^h (e^{A^c t} dt) \cdot B^c, \quad C^d = C^c. \tag{3}$$

Controller implementation: In this work, we assume that a control application, C_i, comprises three sequential software tasks: (i) *Sensor task*, $\tau_{s,i}$, measures the state (using sensors) of the physical system. (ii) *Control task*, $\tau_{c,i}$, computes the control input based on the system state. (iii) *Actuator task*, $\tau_{a,i}$, applies the control input (using an actuator) to the physical system. These tasks are often mapped on different ECUs due to a physically distributed topology of sensors and actuators. Without loss of generality, we assume that three tasks are mapped on different ECUs. The sensor values measured by $\tau_{s,i}$ are sent on the bus via a message $m_{s,i}$ and the control input is sent as a message $m_{c,i}$. The time between the start of the sensor task and the completion of the actuator task is defined as the *sensor-to-actuator delay*, denoted by d. As shown in Fig. 2, this delay depends on the interplay between the task and message schedules.

Controller design: In this paper, we use the controller model from [19] where the control input u[k] is calculated based on the state at the $(k - \lfloor \frac{d}{h} \rfloor)$-th sampling instant. The mathematical model for the discrete-time delayed system can be written as follows:

$$x[k+1] = A^d x[k] + B^{d,0}u[k] + B^{d,1}u[k-1], \quad y[k] = Cx[k],$$
$$\text{where:} \;\; B^{d,0} = \int_0^{h-d'} (e^{A^c t}dt) \cdot B, \quad B^{d,1} = \int_{h-d'}^{h} (e^{A^c t}dt) \cdot B. \tag{4}$$

Here, $d' = d - \left\lfloor \frac{d}{h} \right\rfloor \cdot h$. Furthermore, we consider the case where the task and message schedules lead to one sampling period sensor-to-actuator delay, i.e., $d = h$, as shown in Fig. 2. Thus, Eq. (4) can be rewritten as follows:

$$x[k+1] = A^d x[k] + B^{d,0}u[k], \quad \text{where:} \;\; B^{d,0} = \int_0^{h} (e^{A^c t}dt) \cdot B. \tag{5}$$

For our assumption of $d = h$, we consider that the control input $u[k]$ is given by:

$$u[k] = Kx[k-1] + Fr, \tag{6}$$

where K and F are the feedback and the feedforward gains respectively, and r represents the reference value that $y[k]$ should eventually reach. The design of a feedback controller involves finding the values for feedback and feedforward gains for a given value of sampling period such that the closed-loop system is stable and the control performance is optimal. Here, we consider a state-feedback controller and we assume that we have the full state information, i.e., the system is fully observable. However, the co-design technique studied in this paper can be trivially extended to output-feedback controllers, e.g., proportional-integral-derivative (PID) controllers, by employing an appropriate control design technique.

Control performance: There are different metrics to measure the closed-loop performance of a controller. Here, we consider two common metrics to measure the control performance J. (i) We study a *quadratic cost function* [39] for which the control performance J can be written as follows:

$$J = \sum_{k=0}^{n=\frac{T_G}{h}} \left(\lambda u[k]^2 + (1-\lambda)\sigma[k]^2\right) \cdot h, \tag{7}$$

where λ is a weight taking the value between 0 and 1, $u[k]$ is the control input and $\sigma[k] = |r - y[k]|$ is the tracking error. Note that we can also add other cost functions, e.g., $J = \int_0^{T_G} \left[\lambda u(t)^2 + (1 - \lambda)\sigma(t)^2\right] dt$ in the poposed toolchain. The co-design approach used in the toolchain is independent of the choice of the cost function. In Eq. (7), the value of λ should be chosen based on the design requirement. For example, if we want the system output to stabilize at the reference value quickly by spending more energy, we will choose a lower value of λ. Here, we multiply the cost for each discrete step by the sampling period h which is different from the quadratic cost usually considered in the literature. This is required because we want to compare controllers designed for different sampling periods based on this metric. Therefore, we calculate the quadratic cost until a certain given time T_G from which the number of samples n for a given sampling period h can be calculated as T_G/h. (ii) We also consider the *settling time* to evaluate the control performance, i.e., $J = \xi$, where ξ denotes the time necessary for the system to reach and remain within 1% of the reference value.

For both metrics, we assume that the initial condition and the reference input are in the design specification. It is challenging to design a controller that gives optimal performance for different combinations of these values, and hence, it is reasonable to assume that these values are selected as per requirements. Note that we evaluate the control performance considering a negligible system noise. Depending on the control requirements, one of the aforementioned performance metrics can be used. For both metrics, a smaller value of J implies a better control performance. Each application C_i with a control performance J_i must satisfy a certain requirement J_i^r (i.e., $J_i \leq J_i^r$) as given in the design specification. In a system consisting of multiple control applications with

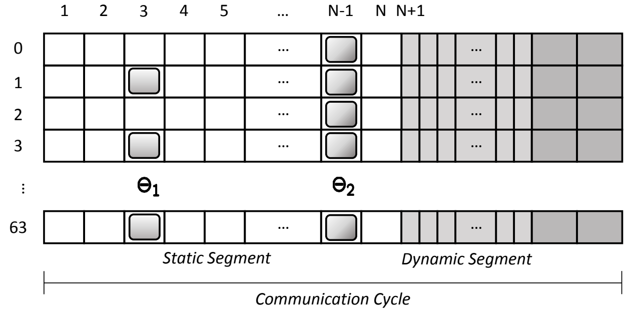


Fig. 3. An example of FlexRay schedules.

different plant models and performance metrics, we need to normalize the control performance in order to compare and combine them. Here, we normalize the control performance as follows:

$$J_i^n = \frac{100\% \cdot J_i}{J_i^r}. \tag{8}$$

Thus, the overall QoC for a set of control applications C can be represented as a weighted sum of the normalized values as follows:

$$J_o = \sum_{C_i \in C} w_i J_i^n, \tag{9}$$

where w_i implies the relative importance of each control application and $\sum w_i = 1$. For example, if it is equally important to optimize the control performance of each application, $C_i \in C$, we get the weights as $w_i = \frac{1}{|C|}$.

2.2 FlexRay-Based ECU Networks

ECU task model: We consider that the real-time operating system on an ECU runs a time-triggered non-preemptive scheduling scheme. On an ECU E_k, a set of periodic tasks, denoted by $\mathcal{T}_{E_k}$, are mapped. The schedule for a task $\tau_{x,i}$ can be defined by a tuple $\{p_{x,i}, o_{x,i}, e_{x,i}\}$, where $p_{x,i}$, $o_{x,i}$ and $e_{x,i}$ denote respectively the period, the offset and the worst-case execution time (WCET) of the task. Note that $x \in \{s, c, a\}$ denotes the type of task, i.e., sensor, control or actuator task. We denote $t(\tau_{x,i}, k)$ and $\tilde{t}(\tau_{x,i}, k)$ as the starting and the latest completion time of the k-th ($k \in \mathbb{Z}^*$) instance of a task $\tau_{x,i}$, which are given by:

$$t(\tau_{x,i}, k) = o_{x,i} + kp_{x,i}, \qquad \tilde{t}(\tau_{x,i}, k) = o_{x,i} + kp_{x,i} + e_{x,i}. \tag{10}$$

We also consider a set of *communication tasks* besides the application tasks. The communication task on the sending ECU writes the data produced by the application task into the corresponding *transmit buffer* of the communication controller, and on the receiving ECU, it reads the data from the corresponding *receiver buffer* and forwards them to the application task. The nature of these communication tasks depends on the specific implementation. Here, we consider that the execution time of all communication tasks is upper-bounded by ϵ. We schedule communication tasks immediately (i) after a task that sends data and (ii) before a task that receives data.

Communication model: FlexRay [14] is an automotive communication protocol usually used by safety-critical applications. It allows both time-triggered (TT) and event-triggered (ET) communication. FlexRay communication is organized as a series of *cycles* where each cycle has a length denoted as T_{bus}. Each communication cycle contains mainly: the *static segment* (ST) and the *dynamic segment* (DYN), where TDMA and Flexible TDMA (FTDMA) communication services are implemented respectively. In this work, we study transmission of messages only on the static segment of FlexRay. The static segment is split into a number of *static slots* of equal length Δ. Here, we represent the slots on the static segment as $\mathcal{S} = \{1, 2, ..., N\}$. Once a static slot is assigned, if no data is sent in a specific communication cycle, the static slot will still be occupied.

We consider the case where a sequence of 64 cycles repeats infinitely. In a sequence, each communication cycle is indexed by a *cycle counter* that counts from 0 to 63 and is then reset to 0. The schedule of a FlexRay frame Θ_i can be defined by a tuple (S_i, B_i, R_i), where S_i represents the slot number, B_i represents the *base cycle*, and R_i is the *repetition rate*. The repetition rate is the number of communication cycles that elapse between two consecutive transmissions of the same frame and takes the value $R_i \in \{2^n | n \in \{0, ..., 6\}\}$. The base cycle is the offset of the cycle counter, i.e., it is the cycle where the frame is scheduled for the first time. The sequence of 64 communication cycles and a few examples of FlexRay schedules are shown in Fig. 3. In the context of this work, we consider the FlexRay versions 2.1 and 3.0.1. In the later version, *slot multiplexing* amongst different ECUs is allowed, i.e., a particular slot S_i can be assigned to different ECUs in different communication cycles. However, this is not allowed in the former version. We further assume that each FlexRay frame, Θ_i, is packed with only one message, m_i. The start and the completion time of the k-th instance ($k \in \mathbb{Z}^*$) of a FlexRay frame (Θ_i) transmission, which are denoted respectively as $t(\Theta_i, k)$ and $\tilde{t}(\Theta_i, k)$, can be written as follows:

$$t(\Theta_i, k) = B_i T_{bus} + k R_i T_{bus} + (S_i - 1)\Delta, \quad \tilde{t}(\Theta_i, k) = B_i T_{bus} + k R_i T_{bus} + S_i \Delta. \tag{11}$$

In this paper, we consider the bus resource usage as the fraction of bandwidth in the static segment that is allocated to the control applications. This can be translated into the percentage of static slots assigned in every sequence of 64 communication cycles. Let Γ denote the set of all FlexRay frames that will be sent on the static segment, where $\Theta_i \in \Gamma$, then the resource usage U can be written as follows:

$$U = \frac{100\%}{64N} \sum_{\Theta_i \in \Gamma} \frac{64}{R_i} = \frac{100\%}{N} \sum_{\Theta_i \in \Gamma} \frac{1}{R_i}, \tag{12}$$

where, the smaller the value of U is, the better is the resource usage, i.e., more bandwidth can be assigned to non-control applications. U can take only a finite number of discrete values because the number of static slots used by the control applications is a natural number less than or equal to $64N$. We can further constrain the value of U using the requirements on the control performance that we will see in Sec. 3.2.3.

2.3 Conventional Design Flow

Typically, for the setting under study, the conventional design methodology only synthesizes the control and platform parameters while respecting the system constraints, e.g., performance and schedulability constraints. For a system with a set of control applications, C, the parameter synthesis boils down to finding for each control application, C_i, (i) the control parameters (including control gains and sampling period) and (ii) the platform parameters (including task and message schedules). The set of design parameters for C_i is denoted by $par_i = par_i^s \cup par_i^c$, where $par_i^c = \{h_i, K_i, F_i\}$ represents the control parameters and $par_i^s = \{o_{s,i}, p_{s,i}, o_{c,i}, p_{c,i}, o_{a,i}, p_{a,i}, S_{s,i}, B_{s,i}, R_{s,i}, S_{c,i}, B_{c,i}, R_{c,i}\}$ captures the platform parameters. Note that the WCETs of the tasks are assumed to be known. The set of design parameters for the whole system is denoted by $\mathcal{P}$, where $\mathcal{P} = \bigcup_{C_i \in C} \; par_i$.

To develop the software for a set of distributed control applications implemented over a FlexRay-based ECU network, the systems and the control engineers usually start with a system specification. On the platform side, these include: (i) ECUs and their hardware and operating system characteristics; (ii) the basic parameters of the FlexRay cluster, e.g., the length of a communication cycle (T_{bus}), the length (Δ) and the number (N) of static slots; and (iii) the task partitions ($\tau_{s,i}, \tau_{c,i}, \tau_{a,i}$) and their mapping $\mathcal{T}_{E_k}$. On the control side, for each application C_i, the plant model ($\{A_i^c, B_i^c, C_i^c\}$)

and the performance requirement (J_i^r) need to be specified. Based on the specification, engineers can design and implement the system in the following three phases.

Design phase: In this phase, system parameters are synthesized and validated based upon some theoretical model of the underlying system. As a first step, the control and the systems engineers negotiate and agree on certain constraints on fundamental parameters like sampling periods and sensor-to-actuator delays for the control applications depending on the platform architecture. For example, ECUs running OSEK/VDX operating system offer only a predefined set of sampling periods. Similarly, the sensor-to-actuator delay of a controller is constrained by the non-negligible time taken by the software tasks running on ECUs and data transmitted over the communication bus which, in turn, depends on the processor speed and memory architecture of the ECUs and the communication protocol and bandwidth of the bus.

Now, the control engineer tries to determine the control gains, sampling period and sensor-to-actuator delay for each application separately based on the plant model and satisfying the constraints on the sampling period and sensor-to-actuator delay. For a state-feedback controller, given a sampling period, the control engineer can use the standard pole-placement technique to calculate the values of control gains that will guarantee the stability of the system [13]. Modeling and simulation tools like Simulink might be used to develop the plant and the controller models for an application. To evaluate the quality of control (e.g., settling time, overshoot, and robustness to noise), closed-loop simulations of the controller and the plant are performed. It might be also required to search the design space for closed-loop poles, sampling period, and sensing-to-actuation delay to ensure that the designed controller meets the control requirements (e.g., quadratic cost and settling time). Finally, the parameter values that meet the control requirement are chosen as the design configuration.

Next, the systems engineer partitions the model of each controller into several software tasks and map those tasks onto ECUs depending on the layout of the physical system, e.g., the placement of the sensors and actuators [43, 48]. Based on the task partitioning and mapping, the requirement on the platform side is to generate valid schedules for the tasks as well as the messages between communicating tasks. This schedule synthesis problem can be formulated as a CSP while considering constraints on designed values of sampling periods and delays, data dependencies, non-overlapping tasks and messages, and other architectural constraints. Mathematical formulations of these constraints are provided in Sec. 3.2.3. The CSP, thus formulated, can be solved using a commercial solver like Gurobi, CPLEX, or Z3. In case no feasible schedule exists, the systems engineer may inform the control engineer to re-design the controllers with re-negotiated constraints on sampling periods and delays. As a result, this conventional design paradigm can be iterative and time-consuming. Moreover, in order to save time, if the engineers on either side make conservative assumptions (e.g., use more ECUs) then the design becomes resource-inefficient which may not be sustainable in the cost-sensitive automotive domain [17, 25]. Furthermore, this conventional design involves significant manual intervention, and therefore, can be error-prone and tedious.

Software implementation phase: In this phase, the complete system software is modeled. The Simulink models of the controllers, as developed by the control engineer in the design phase are combined to form the software model. Subsequently, keeping the control laws intact, this model is further manually modified to incorporate: (i) the architecture model, i.e., ECU and FlexRay parameters; (ii) implementation-specific details like task partitioning and mapping, data mapping and frame packing; and (iii) application-specific details, e.g., depending on the type of speed encoder (i.e., absolute or incremental), the sensor task needs to be modeled differently. This updated system software is then manually configured according to the task and message schedules obtained from the design phase. Here, SIMTOOLS provides specific blocksets that enable modeling of the FlexRay network and ECUs, partitioning and mapping of tasks, packing of data or signals into frames, and configuration of task and message schedules. After the software implementation, simulations are typically run to validate the correctness of the implemented models. For example, SIMTOOLS offers a simulation option to validate both functionality and timing correctness.

Code generation and hardware implementation phase: In this phase, the binary file for each ECU is generated from the software model and then flashed. First, the complete model developed so far is split into separate models where each represents the software that will run on an ECU. Next, C-code and, subsequently, the binary file are generated for each ECU from the corresponding software model. Here, for example, SIMTOOLS offers a function *Split and Build* that automates the software partitioning where the software is developed in Simulink using SIMTOOLS/SIMTARGET toolboxes. It is then possible to generate C-code and the binary file for each ECU by invoking Simulink Real-Time Workshop (RTW) together with SIMTARGET. Simulink RTW facilitates code generation for Simulink blocks while SIMTARGET generates codes for SIMTOOLS/SIMTARGET blocks. The binary files, thus generated, are then used to flash the ECUs for further hardware-in-the-loop (HIL) testing.

The aforementioned development process has several disadvantages:

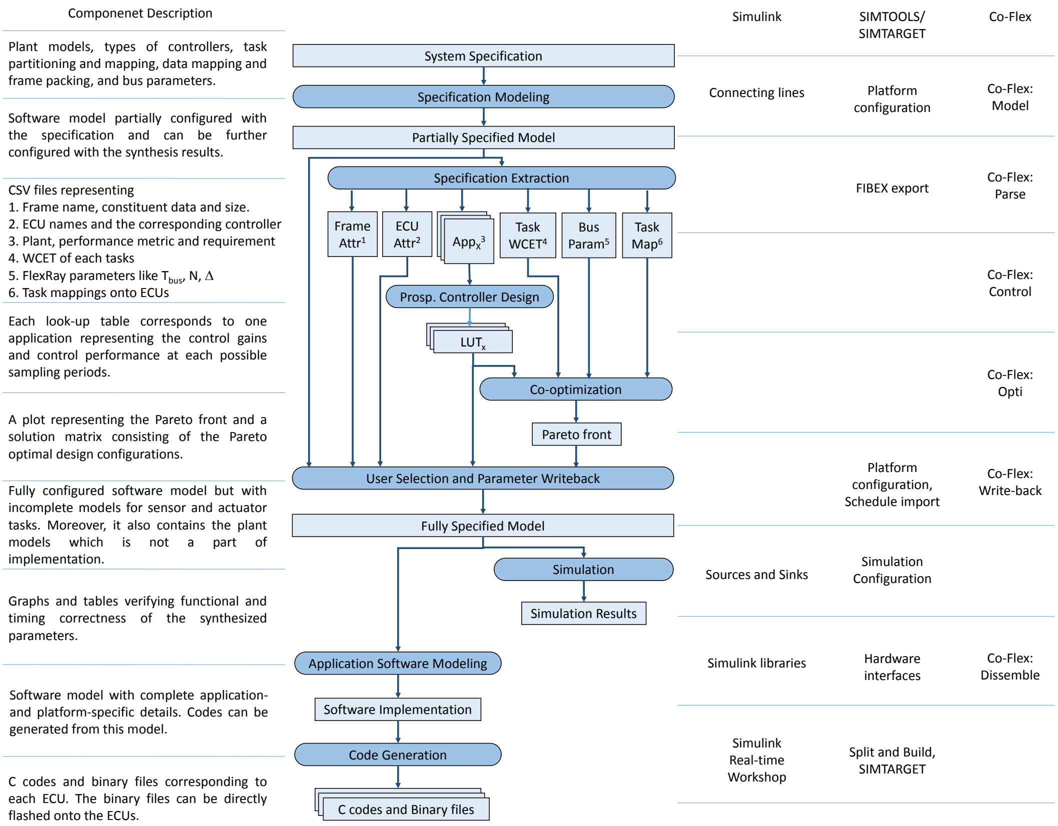
- Developers need to model the system from scratch, which is time-consuming.
- Developers manually formulate the control and the platform design problems from the system specification.
- In the software implementation phase, the model is manually configured with the values of the platform parameters that are obtained from the design phase, which is error-prone, time-consuming and cumbersome.
- The final implementation might not be efficient with respect to (i) the QoC of the applications and (ii) the amount of computation and communication resources that are allocated to them.

3 PROPOSED DESIGN FLOW AND TOOLCHAIN

In this paper, we propose a new design flow for distributed embedded controllers as shown in Fig. 4. We have also used it for developing FlexRay-based automotive control software. We have further developed a toolchain, named *Co-Flex*, to support software development based on the proposed methodology in MATLAB/Simulink environment in conjunction with SIMTOOLS/SIMTARGET toolboxes. Using the SIMTOOLS and SIMTAGET toolboxes require a license that is only available in an USB stick (i.e., they are in the form of license dongles). Without these toolboxes, Co-Flex cannot be instantiated. This, unfortunately, restricts the use of our toolbox in the public domain because of the dependency on the commercial SIMTOOLS and SIMTAGET toolboxes. Hence, we are not making our toolbox publicly available. This shortcoming is inherent in any toolchain that is targeted towards industrial use and relies on commercially-available design tools. For the future, we plan to build a similar toolchain using public domain software tools and make it available for the academic community. But even without it, we believe that this paper – describing how industry-strength tools can be combined for the purpose of control-platform co-design – is useful for the academic community too.

The proposed design flow along with Co-Flex can overcome the disadvantages of the conventional approach in the following ways.

- Co-Flex provides a library that assists developers to conveniently model a controller with implementation-specific details like task partitioning and mapping, task schedules, data mapping, and frame packing. Thus, it is not required to develop the software model from scratch and, instead, Co-Flex blocks can be used and parameterized according to the specification.



Componenet Description

Plant models, types of controllers, task partitioning and mapping, data mapping and frame packing, and bus parameters.

Software model partially configured with the specification and can be further configured with the synthesis results.

CSV files representing
1. Frame name, constituent data and size.
2. ECU names and the corresponding controller
3. Plant, performance metric and requirement
4. WCET of each tasks
5. FlexRay parameters like T_{bus}, N, Δ
6. Task mappings onto ECUs

Each look-up table corresponds to one application representing the control gains and control performance at each possible sampling periods.

A plot representing the Pareto front and a solution matrix consisting of the Pareto optimal design configurations.

Fully configured software model but with incomplete models for sensor and actuator tasks. Moreover, it also contains the plant models which is not a part of implementation.

Graphs and tables verifying functional and timing correctness of the synthesized parameters.

Software model with complete application- and platform-specific details. Codes can be generated from this model.

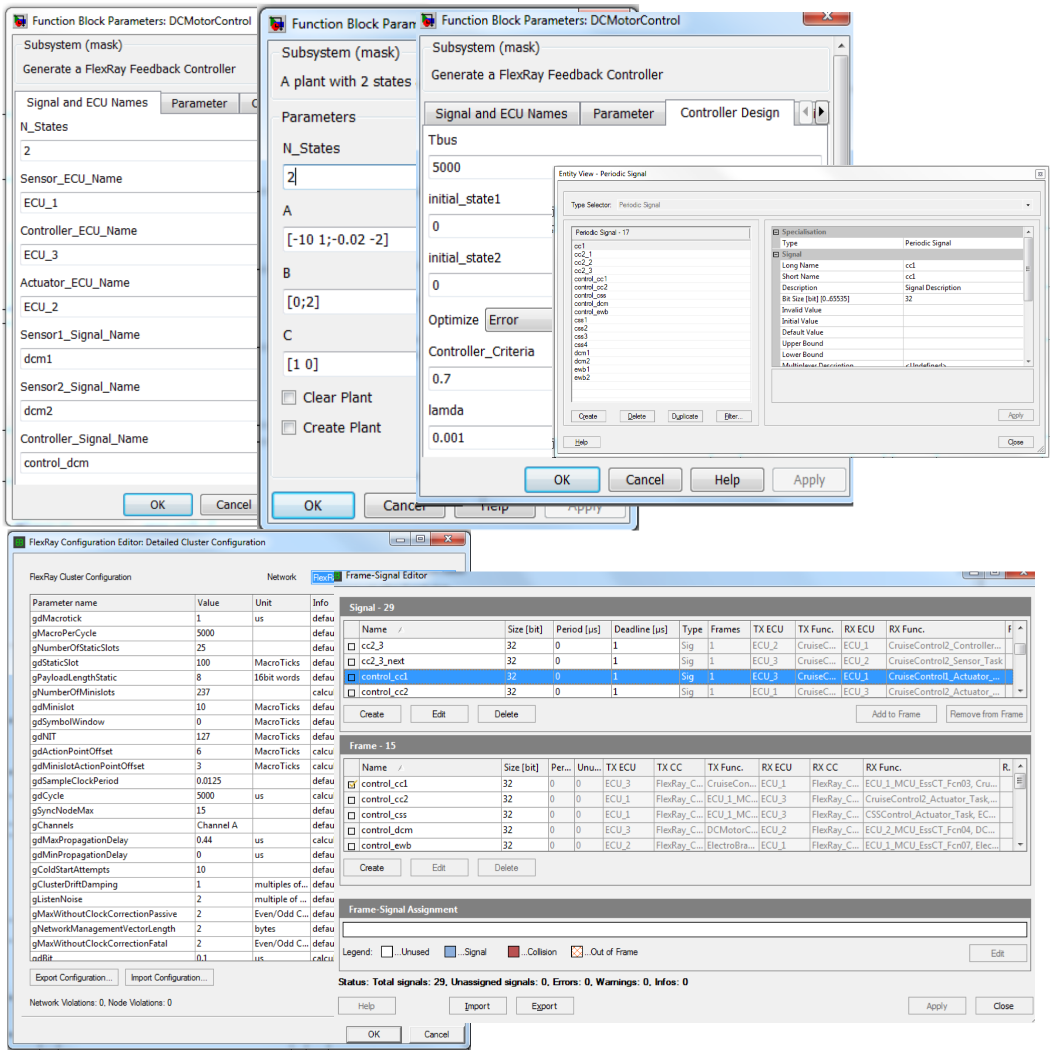
C codes and binary files corresponding to each ECU. The binary files can be directly flashed onto the ECUs.

Simulink	SIMTOOLS/ SIMTARGET	Co-Flex
Connecting lines	Platform configuration	Co-Flex: Model
	FIBEX export	Co-Flex: Parse
		Co-Flex: Control
		Co-Flex: Opti
	Platform configuration, Schedule import	Co-Flex: Write-back
Sources and Sinks	Simulation Configuration	
Simulink libraries	Hardware interfaces	Co-Flex: Dissemble
Simulink Real-time Workshop	Split and Build, SIMTARGET	

Fig. 4. The Proposed Design and Implementation Flow and The Toolchain Support.

- Co-Flex offers a toolbox that automates most of the manual parts in the conventional design flow, e.g., specification extraction, problem formulation, and configuration of design parameters.
- Co-Flex enables automated parameter synthesis. It employs a co-design technique that simultaneously synthesizes the controllers and the platform parameters and, in the process, co-optimizes the overall QoC and the bus resource usage. This technique ensures design optimality and also allows developers to make a trade-off between the two design objectives.

The proposed design flow consists of three phases. In phase (1), i.e., the *specification modeling* phase, the developer creates a model according to the system specification. This model comprises, for each application, the controller template with application-level details (i.e., of the controlled plant and the performance requirement) and implementation-specific information (i.e., about task partitioning and mapping, data mapping, and frame packing). In phase (2), i.e., the *design and software implementation* phase, the parameter synthesis problem is first formulated based on the specification model from phase (1), and the problem is then solved to synthesize the control and platform parameters. Further, the specification model is configured with the synthesized parameter values and more application-specific details are added to obtain the complete software model. Phase (3) or the *code generation* phase is exactly the same as in the case of the conventional design flow where C-code and binary files are generated from the software model to be used to flash the ECUs. Note that we have only replaced the first two phases of the conventional design flow with



Function Block Parameters: DCMotorControl

Subsystem (mask)

Generate a FlexRay Feedback Controller

Signal and ECU Names | Parameter

N_States: 2

Sensor_ECU_Name: ECU_1

Controller_ECU_Name: ECU_3

Actuator_ECU_Name: ECU_2

Sensor1_Signal_Name: dcm1

Sensor2_Signal_Name: dcm2

Controller_Signal_Name: control_dcm

Function Block Param

Subsystem (mask)

A plant with 2 states

Parameters

N_States: 2

A: [-10 1;-0.02 -2]

B: [0;2]

C: [1 0]

□ Clear Plant

□ Create Plant

Function Block Parameters: DCMotorControl

Subsystem (mask)

Generate a FlexRay Feedback Controller

Signal and ECU Names | Parameter | Controller Design

Tbus: 5000

initial_state1: 0

initial_state2: 0

Optimize: Error

Controller_Criteria: 0.7

lamda: 0.001

Entity View - Periodic Signal

Type Selector: Periodic Signal

Periodic Signal - 17

cc1
cc2_1
cc2_2
cc2_3
control_cc1
control_cc2
control_css
control_dcm
control_ewb
css1
css2
css3
css4
dcm1
dcm2
ewb1
ewb2

Specialisation	
Type	Periodic Signal
Signal	
Long Name	cc1
Short Name	cc1
Description	Signal Description
Bit Size [bit] [0..65535]	32
Invalid Value	
Initial Value	
Default Value	
Upper Bound	
Lower Bound	
Multiplexer Description	<Undefined>

FlexRay Configuration Editor: Detailed Cluster Configuration

FlexRay Cluster Configuration

Parameter name	Value	Unit	Info
gdMacrotick	1	us	defau
gMacroPerCycle	5000		defau
gNumberOfStaticSlots	25		defau
gdStaticSlot	100	MacroTicks	defau
gPayloadLengthStatic	8	16bit words	defau
gNumberOfMinislots	237		calcul
gdMinislot	10	MacroTicks	defau
gdSymbolWindow	0	MacroTicks	defau
gdNIT	127	MacroTicks	defau
gdActionPointOffset	6	MacroTicks	calcul
gdMinislotActionPointOffset	3	MacroTicks	calcul
gdSampleClockPeriod	0.0125		defau
gdCycle	5000	us	calcul
gSyncNodeMax	15		defau
gChannels	Channel A		defau
gdMaxPropagationDelay	0.44	us	calcul
gdMinPropagationDelay	0	us	defau
gColdStartAttempts	10		defau
gClusterDriftDamping	1	multiples of...	defau
gListenNoise	2	multiple of ...	defau
gMaxWithoutClockCorrectionPassive	2	Even/Odd C...	defau
gNetworkManagementVectorLength	2	bytes	defau
gMaxWithoutClockCorrectionFatal	2	Even/Odd C...	defau
gdBit	0.1	us	calcul

Network Violations: 0, Node Violations: 0

Frame-Signal Editor

Signal - 29

	Name	Size [bit]	Period [µs]	Deadline [µs]	Type	Frames	TX ECU	TX Func.	RX ECU	RX Func.
□	cc2_3	32	0	1	Sig	1	ECU_2	CruiseC...	ECU_1	CruiseControl2_Controller...
□	cc2_3_next	32	0	1	Sig	1	ECU_3	CruiseC...	ECU_2	CruiseControl2_Sensor_Task
□	control_cc1	32	0	1	Sig	1	ECU_3	CruiseC...	ECU_1	CruiseControl1_Actuator_...
□	control_cc2	32	0	1	Sig	1	ECU_1	CruiseC...	ECU_3	CruiseControl2_Actuator_...

Frame - 15

	Name	Size [bit]	Per...	Unu...	TX ECU	TX CC	TX Func.	RX ECU	RX CC	RX Func.
☑	control_cc1	32	0	0	ECU_3	FlexRay_C...	CruiseCon...	ECU_1	FlexRay_C...	ECU_1_MCU_EssCT_Fcn03, Cru...
□	control_cc2	32	0	0	ECU_1	FlexRay_C...	ECU_1_MC...	ECU_3	FlexRay_C...	CruiseControl2_Actuator_Task,...
□	control_css	32	0	0	ECU_1	FlexRay_C...	ECU_1_MC...	ECU_3	FlexRay_C...	CSSControl_Actuator_Task, EC...
□	control_dcm	32	0	0	ECU_3	FlexRay_C...	DCMotorC...	ECU_2	FlexRay_C...	ECU_2_MCU_EssCT_Fcn04, DC...
□	control_ewb	32	0	0	ECU_2	FlexRay_C...	ElectroBra...	ECU_1	FlexRay_C...	ECU_1_MCU_EssCT_Fcn07, Elec...

Frame-Signal Assignment

Legend: □ ...Unused 🟦 ...Signal 🟥 ...Collision ☒ ...Out of Frame

Status: Total signals: 29, Unassigned signals: 0, Errors: 0, Warnings: 0, Infos: 0

Fig. 5. Specification modeling. (From left to right) Top: (i) task mappings and signals (ii) plant models (iii) control specification (iv) signal description. Bottom: (v) FlexRay configuration (vi) data mapping and frame packing.

the specification modeling phase and the design and implementation phase respectively. In this section, we will explain these two phases in detail together with the tools offered by Co-Flex.

3.1 Specification Modeling

In contrast to the conventional design flow where the modeling of the application software starts only after the synthesis of design parameters, the proposed approach starts with adding the specification details into a software model. For the problem setting described in Sec. 2.1 and Sec. 2.2 respectively, the system specification typically includes (i) plant models, (ii) type of controllers to be used, (iii) task partitioning and mapping, (iv) data mapping and frame packing, and (v) FlexRay bus parameters. We assume that the mathematical models of the physical plants are available and they are controlled using state-feedback controllers.

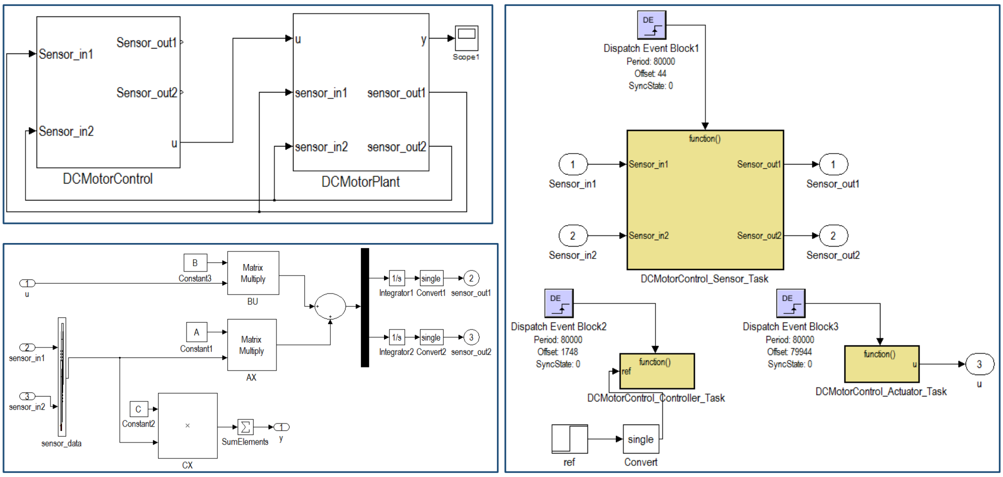


Fig. 6. Partially specified model. (i) (left top) Closed-loop system model (ii) (left bottom) Plant model (iii) (right) Distributed controller implementation.

For task partitioning and mapping, we assume the following to be given: (i) the physical system layout, i.e., the physical distribution of sensors, actuators, and ECUs; and (ii) the network topology, i.e., how the sensors, actuators, ECUs, and FlexRay bus are connected. With this assumption, task partitioning and mapping is partially fixed, e.g., if a sensor is connected to an ECU then it makes sense to map the sensor task on the corresponding ECU. However, the rest can be synthesized by employing any well-established task partitioning and mapping algorithms [23, 40].

Furthermore, from task partitioning and mapping, data dependencies can be derived. The data that needs to be transmitted as messages over the communication bus can be identified. We assume that each frame is packed with only one message. We do not consider frame packing in the co-optimization. We understand that packing a frame with more than one message can result in a more efficient design, however, to ensure the scalability of the co-design scheme, we do not consider this. To the best of our knowledge, existing works on FlexRay frame packing do not consider task and message co-scheduling [26]. Nevertheless, the co-design technique used in this paper can consider the case where it is known apriori how the messages sent from an ECU will be packed into frames. In the future, we can trivially add a rule-based frame packing scheme in Co-Flex. We further assume that FlexRay network parameters like bus cycle time T_{bus}, clock period, number of static slots N, and the size of each slot Δ are fixed based on requirements, e.g., the size of a static slot is determined based on the largest frame in the system.

Now, given the specification, the developer can model a partially configured application software in the Simulink environment using *Co-Flex:Model* library and SIMTOOLS/SIMTARGET. Fig. 5 shows several snapshots on how the specification is modeled using our proposed toolchain conveniently by configuring several block parameters.

Co-Flex: Model – It is a library comprising template blocks that aid the modeling of FlexRay-based control software. For the time being, the library supports modeling of state-feedback controllers only, however, it can be extended in the future for PID, model predictive, and adaptive controllers. The library mainly consists of two template blocks as follows:

- *FeedbackController:* This block accelerates the modeling of a feedback control application that will run on a distributed FlexRay platform. The left block in the top-left snapshot of Fig. 6 is a *FeedbackController*. The parameters to this block include (i) sensor and control message names,

(ii) sensor, control, and actuator task mappings and schedules, (iii) sampling period, (iv) control gains (vii) control performance metric, and (viii) performance requirement. A newly added block basically represents a skeleton implementation of a distributed control application with empty sensor, control, and actuator task models. After parameterizing the block, developers can push a button *create* (provided in the mask) to automatically configure the underlying model to represent a distributed embedded control application as shown in the right snapshot in Fig. 6. Besides, the block is reconfigurable, i.e., developers can use the same block to model a different controller in case of a change in the specification. However, if the order of the corresponding controlled plant is different then developers must first push the button *clear* to go back to the skeleton implementation. Furthermore, a push button *verify* is provided in this block in order to check data consistency between this block and the SIMTOOLS platform configuration block. The inputs of this block are the system states x and the output is the control input u.

- *Plant:* The right block in the top-left snapshot of Fig. 6 is a *Plant*. Developers can specify the plant model by parameterizing this block. It can represent a controlled plant of any order. The parameters to this block include (i) the system order, (ii) the state matrix A^c, (iii) the input matrix B^c, and (iv) the output matrix C^c. After parameterizing the block, developers can push a button *create* to automatically build an underlying plant model as shown in the bottom-left diagram in Fig. 6. This block is also reconfigurable and developers can use the push button *clear* to delete the underlying model. The input of this block is the control input u and the outputs are the system states x and the system output y. Closed-loop system model can be obtained by connecting (i) the outputs x of this block to the inputs of the *FeedbackController* block and (ii) the output u of the *FeedbackController* block to the input of this block. The purpose of this block is twofold: (i) The plant specification can be read from this block to design the controller. (ii) The underlying plant model can be used for closed-loop simulations to validate the designed control and platform parameters before the hardware implementation.

Now, using *Co-Flex:Model* and SIMTOOLS/SIMTARGET, the specification modeling is carried out as follows: (i) For each control application, insert a *FeedbackController* block and a *Plant* block and parameterize them according to the specification. Then, create the models for the plant and the partially-specified controller respectively and connect them as shown in the top-left snapshot in Fig. 6. (ii) Insert a *Database File Block* provided by SIMTOOLS that allows to specify the complete platform configuration. (iii) In the *Database File Block*, configure the message signals, the ECUs, and the FlexRay network according to the specification. (iv) In the *Database File Block*, use *Import constraints from model* feature so that the task and message mappings are read automatically from the model. (v) Configure frames and assign messages to frames such that for each control application there are two FlexRay frames, i.e., a sensor signal frame and a control signal frame respectively. (vi) Finally, the modeling correctness can be verified by pushing the button *verify* in each of the *FeedbackController* blocks.

The output of this phase is a partially-specified software model, as mentioned in Fig. 4. The control and schedule parameters must be written to this model and more application-level details are added to synthesize the complete software model.

3.2 Design and Software Implementation

This phase starts with the extraction of relevant information from the specification model which is required to formulate the parameter synthesis problem. Based on the extracted information, a co-optimization problem is formulated. Here, the co-optimization problem can be defined as to find the set of parameters par_i for each control application, where $par_i \in \mathcal{P}$, while optimizing the total FlexRay bus resource usage as in Eq. (12) and the overall QoC as in Eq. (9).

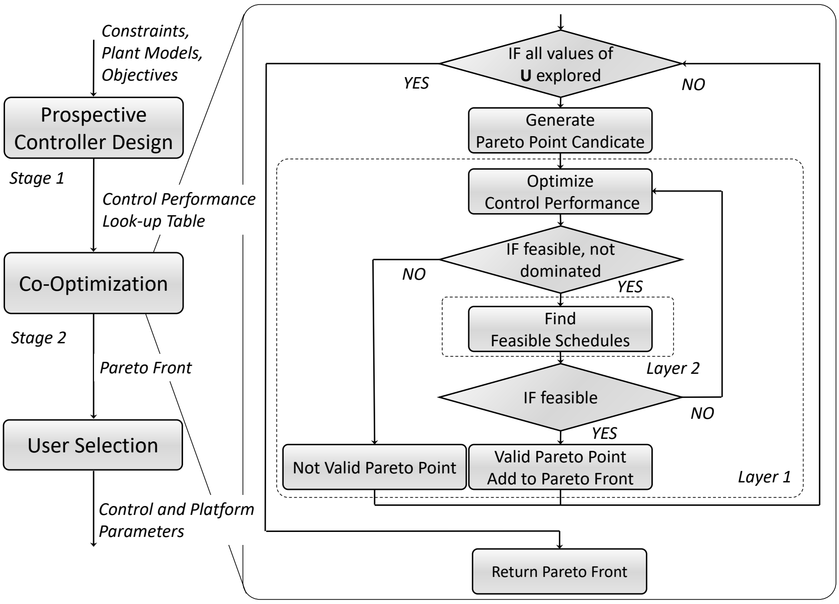

Constraints,
Plant Models,
Objectives
Prospective Controller Design
Stage 1
Control Performance Look-up Table
Co-Optimization
Stage 2
Pareto Front
User Selection
Control and Platform Parameters
IF all values of **U** explored
YES
NO
Generate Pareto Point Candicate
Optimize Control Performance
IF feasible, not dominated
NO
YES
Find Feasible Schedules
Layer 2
IF feasible
NO
YES
Not Valid Pareto Point
Valid Pareto Point Add to Pareto Front
Layer 1
Return Pareto Front

Fig. 7. The co-optimization approach.

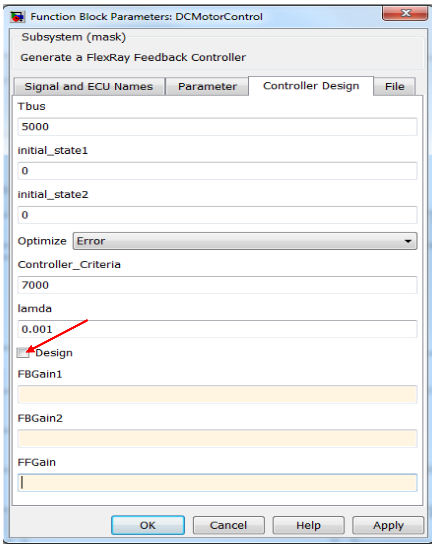
In this work, we implement a hybrid optimization approach [36], as shown in Fig. 7, that can efficiently solve the co-design problem. The optimization comprises two stages. In the first stage, for each control application, prospective controllers are synthesized that optimize the control performance at different sampling periods and the results are recorded in a look-up table. In this stage, we employ the pole placement design method and search the pole space for the optimal controller at each possible sampling period. In the second stage, the co-optimization stage, we synthesize both control and the platform parameters based on the system constraints, objectives and the look-up tables obtained in the first stage. Here, we formulate a bi-objective optimization problem and use a customized approach to generate a Pareto front considering the two objectives.

Based on the obtained Pareto front, the developer can select one of the Pareto points that is the most suitable for the overall design requirements. The parameter values corresponding to the selected Pareto point are then written back into the software model. The software is modeled further to incorporate application-specific details, i.e., the models for sensor and actuator tasks. The software model thus obtained can be simulated using the plant models from the previous phase. Finally, the plant models can be removed and then binary and C-code files can be generated to be used to flash the ECUs.

This phase is essentially divided into five stages, as shown in Fig. 4, that are discussed in detail in the rest of this section.

3.2.1 Specification Extraction.

This is the first stage of the design and software implementation phase and is necessary for the automated formulation of the co-optimization problem. In this stage, developers first export the platform configuration from the specification model as a Field Bus Exchange Format (FIBEX) file. For this, developers use the *Export as FIBEX* feature that is provided in the SIMTOOLS *Database File Block*. FIBEX is a standard format used by the automotive industry to exchange data. A FIBEX file is an Extensible Markup Language (XML) document with a standard XML Schema Definition (XSD). It can be read by any standard parsing tool to extract important information. The exported



Function Block Parameters: DCMotorControl

Subsystem (mask)

Generate a FlexRay Feedback Controller

Tbus: 5000

initial_state1: 0

initial_state2: 0

Optimize: Error

Controller_Criteria: 7000

lamda: 0.001

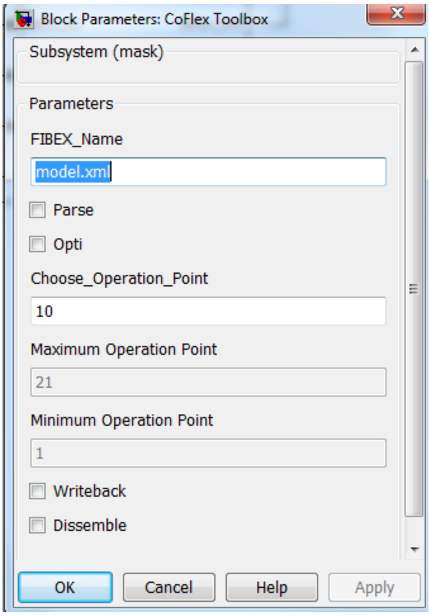
□ Design

FBGain1:

FBGain2:

FFGain:

Fig. 8. Co-Flex: Control tool.



Block Parameters: CoFlex Toolbox

Subsystem (mask)

Parameters

FIBEX_Name: model.xml

□ Parse

□ Opti

Choose_Operation_Point: 10

Maximum Operation Point: 21

Minimum Operation Point: 1

□ Writeback

□ Dissemble

Fig. 9. Parse, Opti, Writeback, Dissemble tools.

FIBEX contains dummy values for task and message schedules as they are not yet calculated. For the design, the relevant information in the FIBEX file includes: (i) the FlexRay parameters like the bus cycle time (T_{bus}), the number of static slots (N), and the length of a static slot (Δ); (ii) the ECU attributes like the names of the ECUs and the corresponding FlexRay controllers; (iii) the task attributes like the task names, WCETs and mappings; and (iv) the FlexRay frame attributes like the names, the data mapping, and the size of the constituent data. However, note that the information is not complete. The obtained FIBEX does not contain any information about the plant model, the performance metric or the performance requirement for an application, while they are necessary to design the controller. Co-Flex toolbox offers a tool, *Parse*, that helps to extract the relevant information from the specification model and store it in a systematic manner.

Co-Flex: Parse – This tool can be invoked from the *Co-Flex Toolbox* block, as shown in Fig. 9, by checking the corresponding checkbox. It basically helps in specification extraction and systematic storage. It parses the specification model as well as the exported FIBEX from the model. This tool uses common MATLAB semantics (e.g., find_system and get_param) to parse the software model and the MATLAB Document Object Model (DOM) parser to parse the FIBEX file. All the required details can be extracted using this tool. Further, this tool organizes the extracted information, as given in Fig. 4, into several easy-to-access csv files, i.e., *BusParam.csv*, *ECUAttr.csv*, *WCET.csv*, *TaskMap.csv*, *FrameData.csv*, and *AppX.csv* (for application *X*). These files can then be accessed by the following stages for the problem formulation as well as when writing back the parameter values into the software model.

3.2.2 *Prospective Controller Design.*

The control performance J_i of an application C_i depends mainly on three factors: (i) the sampling period h_i, (ii) the sensor-to-actuator delay d_i, and (iii) the control gains K_i and F_i. However, we consider that for each application, the task and message schedules lead to the case where the delay is equal to the sampling period, i.e., $d_i = h_i$. This is a reasonable consideration because the impact of the delay on the control performance is insignificant compared to that of the sampling period [28]. Also, this offers more flexibility in scheduling the tasks and messages. This would reduce the dimensions of the design space from all three factors (i) - (iii) to only (i) and (iii), thus reducing the problem complexity. The co-design approach under study can also be applied to other cases, e.g., a constant delay or a delay value proportional to the sampling period.

With the assumption $d_i = h_i$, the purpose of the prospective controller design stage is to determine for each application the control gains that optimize the control performance at each possible value of the sampling period. Here, we make use of the fact that the sampling period can only take a finite number of values to prune the design space further. Considering that each control application C_i is implemented by the tasks $\tau_{s,i}$, $\tau_{c,i}$, $\tau_{a,i}$ and messages $m_{s,i}$, $m_{c,i}$, there is a dependency between the sampling period h_i and the repetition rate of the messages $R_{s,i}$, $R_{c,i}$. This is represented as follows:

$$h_i = R_{s,i} T_{bus} = R_{c,i} T_{bus}. \tag{13}$$

Due to the fact that $R_{s,i}$ and $R_{c,i}$ can only take values in $\{2^k | k \in \{0, ..., 6\}\}$, the choice of h_i becomes constrained accordingly. In this stage, Co-Flex offers a tool *Control* that can be invoked for each application to design prospective optimal controller at each possible sampling period.

Co-Flex: Control – This tool can be invoked from the *FeedbackController* block by checking a checkbox indicated by an arrow in Fig. 8. In order to design prospective controllers for application X, this tool uses *AppX.csv* to fetch the plant model, the control performance metric, and the performance requirement. It also reads *BusParam.csv* to obtain the length of a bus cycle T_{bus} that is required to determine the possible values of the sampling period. With $d_i = h_i$, the closed-loop system experiences one sampling period delay. We use the pole placement method reported in [19] for such a delayed system. This method is applied on the equivalent discrete-time system model. It enables to find the control gains for a set of stable poles and, hence, ensures the stability of the closed-loop system. For the optimal pole placement, to the best of our knowledge, there is no closed-form analytical framework. In this work, we search the design space for poles, although it can be computationally expensive. Note that there exists an optimal control technique [12], i.e., the linear quadratic regulator (LQR), when the control performance is computed using a quadratic cost function. It cannot be directly applied when we want to optimize the settling time. Nevertheless, we can easily add the LQR control design technique to the tool.

For $d_i = h_i$, the control performance can be written as $J_i = f(h_i, K_i, F_i)$, i.e., it is a function of the sampling period and the control gains. When the plant model $\{A_i^c, B_i^c, C_i^c\}$, the sampling period h_i, and the control gains $\{K_i, F_i\}$ are known, the control performance is determined via the closed-loop simulation of the plant and the controller as per Eq. (5) and Eq. (6) respectively. The control performance at each feasible value of the sampling period, i.e., $h_i^k = 2^k T_{bus}$, can also be written as $J_i^k = g(K_i^k, F_i^k)$, i.e., it is a function of control gains only. The control gains can be uniquely computed based on a chosen set of poles, thus, we can write $J_i^k = g'(\rho_i^k)$, where ρ_i^k denotes the set of poles. The *Control* tool searches for the set of poles that optimizes the control performance. Using the method in [19], stable controllers can only be designed for a *restricted* set of poles. We prune the design space accordingly. Note that the search is carried out with a specific granularity in the restricted pole space. Considering that this search discretizes the pole space, it does not guarantee to obtain the optimal controller. However, the smaller the search granularity is, the better are the chances of getting a controller closer to the optimum. We denote that, corresponding to a sampling period h_i^k, the *Control* tool synthesizes the control gains K_i^{k*} and F_i^{k*} that optimize the control performance to a value J_i^{k*}. Interested readers are encouraged to read [36] for the detailed implementation of the search algorithm to design an optimal controller for a given sampling period.

The *Control* tool further formulates a look-up table (LUT) for each control application C_i, as stated in Fig. 4. The LUT stores, for each possible sampling period h_i^k, the optimal control performance J_i^{k*} and the corresponding control gains K_i^{k*} and F_i^{k*}. In the co-optimization stage, the sampling periods of the control applications will be used as variables during the problem formulation. The objective of the overall QoC can, therefore, be formulated as a discrete function of the sampling periods. Corresponding to the values of the sampling periods, the control laws can be selected by referring to these LUTs. Therefore, the sampling periods here serve as the main interface between the prospective controller design stage and the co-optimization stage.

3.2.3 *Co-Optimization.*

With the results from the prospective controller design stage, the co-optimization problem can be formulated as a constrained programming model. The parameters to be synthesized include (i) the sampling periods of the control applications, (ii) the task schedules, and (iii) the message schedules. We consider the platform constraints and the control performance constraint that are formulated as follows:

$$\forall C_i \in \mathcal{C}, x \in \{s,c,a\}, y \in \{s,c\}, \quad p_{x,i} = R_{y,i}T_{bus} = h_i. \tag{14}$$

$$\forall C_i \in \mathcal{C}, x \in \{c,a\}, y \in \{s,c\}, \quad \alpha_{x,i}, \beta_{y,i} \in \{0,1\}. \tag{15}$$

$$\forall C_i \in \mathcal{C}, \quad o_{s,i} + e_{s,i} + \epsilon < (B_{s,i} + \beta_{s,i}R_{s,i})T_{bus} + (S_{s,i} - 1)\Delta. \tag{16}$$

$$\forall C_i \in \mathcal{C}, \quad (B_{s,i} + \beta_{s,i}R_{s,i})T_{bus} + S_{s,i}\Delta < o_{c,i} + \alpha_{c,i}p_{c,i} - \epsilon. \tag{17}$$

$$\forall C_i \in \mathcal{C}, \quad o_{c,i} + \alpha_{c,i}p_{c,i} + e_{c,i} + \epsilon < (B_{c,i} + \beta_{c,i}R_{c,i})T_{bus} + (S_{c,i} - 1)\Delta. \tag{18}$$

$$\forall C_i \in \mathcal{C}, \quad (B_{c,i} + \beta_{c,i}R_{c,i})T_{bus} + S_{c,i}\Delta < o_{a,i} + \alpha_{a,i}p_{a,i} - \epsilon. \tag{19}$$

$$\forall C_i \in \mathcal{C}, \quad (o_{a,i} + \alpha_{a,i}p_{a,i} + e_{a,i}) - o_{s,i} = h_i. \tag{20}$$

$$\begin{gathered}
\forall \quad C_i, C_j \in \mathcal{C}, \quad x,y \in \{s,c,a\}, \quad E_k \in \mathcal{E} \\
\forall \quad \{m \in \mathbb{Z}^* | 0 \le m \le lcm(p_{x,i}, p_{y,j})/p_{x,i}\} \quad , \quad \{n \in \mathbb{Z}^* | 0 \le n \le lcm(p_{x,i}, p_{y,j})/p_{y,j}\} \\
\textbf{if} \quad (\tau_{x,i} \neq \tau_{y,j}) \land (\tau_{x,i}, \tau_{y,j} \in \mathcal{T}_{E_k}) \quad \textbf{then} \\
\Big[\tilde{t}(\tau_{x,i}, m) + \epsilon \cdot \mathbb{1}(x \in \{s,c\}) < t(\tau_{y,j}, n) - \epsilon \cdot \mathbb{1}(y \in \{c,a\}) \\
\textbf{or} \quad \tilde{t}(\tau_{y,j}, n) + \epsilon \cdot \mathbb{1}(y \in \{s,c\}) < t(\tau_{x,i}, m) - \epsilon \cdot \mathbb{1}(x \in \{c,a\})\Big].
\end{gathered} \tag{21}$$

[1]

$$\begin{gathered}
\forall \quad C_i, C_j \in \mathcal{C}, \quad x,y \in \{s,c\} \\
\forall \quad \{n \in \mathbb{Z}^* | 0 \le n < max(R_{x,i}, R_{y,j})/R_{x,i}\} \quad , \quad \{m \in \mathbb{Z}^* | 0 \le m < max(R_{x,i}, R_{y,j})/R_{y,j}\} \\
\textbf{if} \quad (\Theta_{x,i} \neq \Theta_{y,j}) \land (S_{x,i} = S_{y,j}) \quad \textbf{then} \quad B_{x,i} + nR_{x,i} \neq B_{y,j} + mR_{y,j}.
\end{gathered} \tag{22}$$

$$\forall \ C_i \in \mathcal{C}, \ x \in \{s,c\}, \ 1 \le S_{x,i} \le N. \tag{23}$$

$$\forall \ C_i \in \mathcal{C}, \ x \in \{s,c\}, \ 0 \le B_{x,i} < R_{x,i}. \tag{24}$$

$$\forall \ C_i \in \mathcal{C}, \ x \in \{s,c,a\}, \ 0 \le o_{x,i} < p_{x,i}. \tag{25}$$

$$\forall k \in \{0,1,...6\}, \quad J_i^{k*} \le J_i^r \iff h_i^k \in dom[h_i] \quad \text{and} \quad h_i == 2^k T_{bus} \iff J_i == J_i^{k*}. \tag{26}$$

Constraint (14) implies that the sampling period of an application must be equal to the period of repetition of the component tasks and messages. Constraints (15) to (19) state that in a control application all task executions and message transmissions have to be carried out in the correct

[1] $\mathbb{1}(.)$ takes the value of 1 if the input is true and 0 if otherwise.

temporal order, i.e., certain precedence relations must be followed. Constraint (20) is due to the assumption of one sampling period sensor-to-actuator delay. Constraints (21) and (22) lay down respectively that no two tasks or two messages mapped on the same resource must overlap in time. Constraints (23) and (24) state the permissible values of slot ids and base cycles respectively. Constraint (25) asserts that each time window of duration equal to the task period must have at least one task instance. Constraint (26) implies that the sampling period of an application can attain only those values corresponding to which the designed optimal controller from the previous stage satisfies the performance requirement. Furthermore, in case of FlexRay 2.1, an additional constraint must be considered as slot multiplexing between different ECUs is not allowed. This is given by:

$$\begin{gathered}\forall\ C_i, C_j \in \mathcal{C},\ \ x, y \in \{s, c\},\ \ \exists E_k \in \mathcal{E},\\ \textbf{if}\quad (\Theta_{x,i} \neq \Theta_{y,j}) \wedge (\tau_{x,i} \in \mathcal{T}_{E_k}) \wedge (\tau_{y,j} \notin \mathcal{T}_{E_k}) \quad \textbf{then} \quad S_{x,i} \neq S_{y,j}.\end{gathered} \tag{27}$$

Moreover, the optimization objectives of bus resource usage and overall QoC can be formulated respectively as follows:

$$U = \frac{100\%}{64N} \sum_{C_i \in \mathcal{C}} \left(\frac{64}{R_{s,i}} + \frac{64}{R_{c,i}}\right) = \frac{100\%}{64N} \sum_{C_i \in \mathcal{C}} \frac{128 T_{bus}}{h_i}. \tag{28}$$

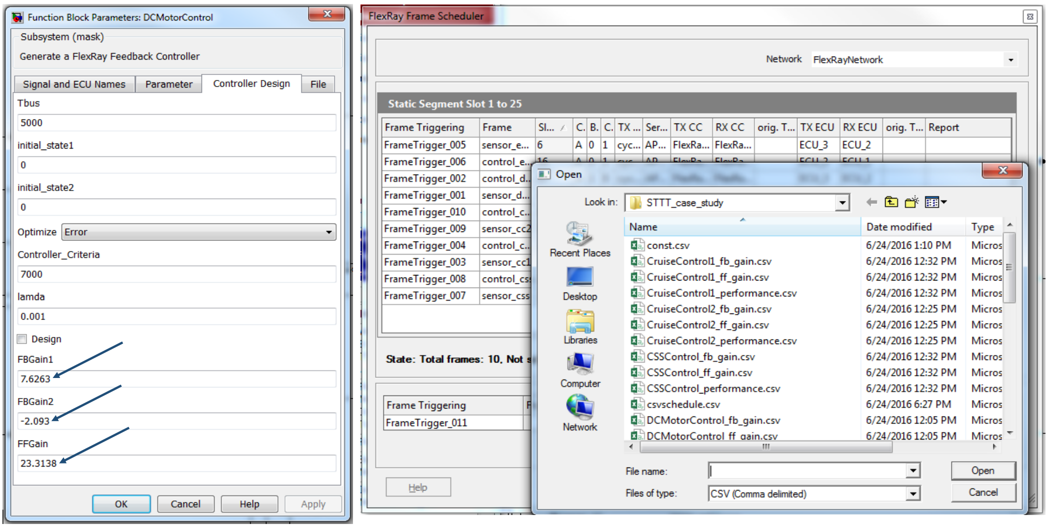
$$J_o = \sum_{C_i \in \mathcal{C}} w_i J_i^n = \sum_{C_i \in \mathcal{C}} w_i \sum_k \mu_{i,k} J_i^{k*(n)}, \quad \text{where} \sum_k \mu_{i,k} = 1 \text{ and } J_i^{k*(n)} = \frac{100 J_i^{k*}}{J_i^r}. \tag{29}$$

As formulated above, the control-platform co-design for the setting under study can be modeled as a constrained optimization problem with two objectives. There exist several methods dealing with multi-objective optimization. A simple way is to convert the multiple objectives into one single objective with scalarization. However, the problems using this method here are as follows: (i) The two objectives are completely different in nature and it is challenging to combine them as a single metric. (ii) It would not be possible for developers to understand the design trade-off, which is necessary because the two objectives are noticed to be often conflicting. In this case, a more informative and developer-friendly approach is to first generate the Pareto front and let developers explore the trade-off between the two objectives according to their customized preference. Co-Flex toolbox comprises a tool named *Opti* that can be used to automate the formulation of the co-optimization problem and, thereafter, for finding the Pareto-optimal solutions to the problem.

Co-Flex: Opti – This tool can be invoked from the *Co-Flex Toolbox* block, as shown in Fig. 9, by checking the corresponding checkbox. It can automatically formulate the problem according to the constraints and objectives in Eqs. (14) – (29) by referring to the files generated in the *specification extraction* stage and the look-up tables from the *prospective controller design* stage. In order to solve the problem and obtain the desired Pareto front, the tool uses a customized optimization approach. Considering that the resource usage U only takes a limited number of values, it first computes the maximum and the minimum possible values of the resource usage U^+ and U^- as follows:

$$U^+ = \frac{100}{64N} \sum_{C_i \in \mathcal{C}} \frac{128 T_{bus}}{\max\limits_{h_i \in dom[h_i]} (h_i)}, \qquad U^- = \frac{100}{64N} \sum_{C_i \in \mathcal{C}} \frac{128 T_{bus}}{\min\limits_{h_i \in dom[h_i]} (h_i)}. \tag{30}$$

For each possible value of U from U^- to U^+, i.e., given the equality constraint on U, it solves the optimization problem with J_o as the single objective and obtain a solution. The additional constraint is that J_o of this solution has to be better than J_o of the last solution, in order to ensure that all solutions are non-dominated. The co-optimization problem with two objectives is, thus, turned into a series of single-objective optimization problems, where each may generate a Pareto point. Here, popular approaches like Mixed Integer Linear Programming (MILP) or meta-heuristic methods cannot be applied directly to solve each of the single-objective optimization problems.


Function Block Parameters: DCMotorControl

Subsystem (mask)

Generate a FlexRay Feedback Controller

Tbus: 5000

initial_state1: 0

initial_state2: 0

Optimize: Error

Controller_Criteria: 7000

lamda: 0.001

□ Design

FBGain1: 7.6263

FBGain2: -2.093

FFGain: 23.3138

FlexRay Frame Scheduler

Network: FlexRayNetwork

Static Segment Slot 1 to 25

Frame Triggering	Frame	Sl...	C.	B.	C.	TX ...	Ser...	TX CC	RX CC	orig. T...	TX ECU	RX ECU	orig. T...	Report
FrameTrigger_005	sensor_e...	6	A	0	1	cyc...	AP...	FlexRa...	FlexRa...		ECU_3	ECU_2		
FrameTrigger_006	control_e...	16	A	0	1	cyc...	AP...	FlexRa...	FlexRa...		ECU_2	ECU_1		
FrameTrigger_002	control_d...													
FrameTrigger_001	sensor_d...													
FrameTrigger_010	control_c...													
FrameTrigger_009	sensor_cc2													
FrameTrigger_004	control_c...													
FrameTrigger_003	sensor_cc1													
FrameTrigger_008	control_cs													
FrameTrigger_007	sensor_css													

State: Total frames: 10, Not s

Frame Triggering	F
FrameTrigger_011	

Open

Look in: STTT_case_study

Name	Date modified	Type
const.csv	6/24/2016 1:10 PM	Micros
CruiseControl1_fb_gain.csv	6/24/2016 12:32 PM	Micros
CruiseControl1_ff_gain.csv	6/24/2016 12:32 PM	Micros
CruiseControl1_performance.csv	6/24/2016 12:32 PM	Micros
CruiseControl2_fb_gain.csv	6/24/2016 12:25 PM	Micros
CruiseControl2_ff_gain.csv	6/24/2016 12:25 PM	Micros
CruiseControl2_performance.csv	6/24/2016 12:25 PM	Micros
CSSControl_fb_gain.csv	6/24/2016 12:32 PM	Micros
CSSControl_ff_gain.csv	6/24/2016 12:32 PM	Micros
CSSControl_performance.csv	6/24/2016 12:32 PM	Micros
csvschedule.csv	6/24/2016 6:27 PM	Micros
DCMotorControl_fb_gain.csv	6/24/2016 12:05 PM	Micros
DCMotorControl_ff_gain.csv	6/24/2016 12:05 PM	Micros

File name:

Files of type: CSV (Comma delimited)

Fig. 10. Writing back the gains (left) and the frame schedules (right).

Considering that some decision variables only appear in the constraints, but are not related to the objectives, a nested two-layer technique, as depicted in Fig. 7, is employed to solve each of the problems. In Layer 1, the outer layer, *Opti* considers only Constraint (26) and an equality constraint on bus usage U derived from Constraint (28). It optimizes the overall QoC as given in Eq. (29). Decision variables related to the objectives, i.e., the sampling periods, are determined in this layer by solving an integer linear programming problem.

In Layer 2, the inner layer, the remaining decision variables are synthesized satisfying the constraints in Eqs. (14) – (25) while substituting the values of sampling periods based on the results of Layer 1. Here, an MILP model is developed for the time-triggered scheduling problem similar to [51]. This process is iterative, i.e., if the synthesis fails in Layer 2, the algorithm goes back to Layer 1 for the next best solution until the Pareto criterion is satisfied. This optimization technique ensures optimality and is also efficient. Details of the optimization technique can be found in [36].

3.2.4 Parameter Writeback.

The Pareto front, that is obtained from the co-optimization stage, consists of a number of Pareto points where each represents a feasible Pareto-optimal solution of the co-optimization problem. The developer can choose one of them for the implementation based on the design requirements. The solution corresponding to the selected Pareto point must be interpreted correctly to represent a valid design configuration. Accordingly, the specification model must be then parameterized. To facilitate convenient write back of the synthesized parameter values in the software model, Co-flex offers a tool *Writeback*.

Co-Flex: Writeback – This tool can be invoked from the *Co-Flex Toolbox* block, as shown in Fig. 9, by checking the corresponding checkbox. It automates (i) completely the result interpretation and (ii) partially the result write back. The solution to the co-optimization problem is stored as a matrix where each row represents a Pareto point. Therefore, if the developer selects the n-th Pareto point then the *Writeback* tool will extract row n from the solution matrix. Thereafter, the task and message schedules can be calculated by this tool through correct interpretation of the elements of the extracted row. The interpretation is done by exploiting the knowledge of problem formulation from the previous stage. That is, if a design parameter is represented by the decision variable *i* then the synthesized value of the parameter is given by element *i* in the extracted row. Furthermore, as depicted in Fig. 4, the values of the control gains can be obtained by this tool from the LUTs generated in the prospective controller design stage according to the synthesized values of the sampling periods.

Following result interpretation, developers need to configure the software model with the obtained parameter values. Towards this, the *Writeback* tool can directly write the control gains (as shown by the pointers in Fig. 10) and the application task schedules in the model automatically. However, for the FlexRay frame schedules, the tool generates a csv file in a format that can be directly imported in SIMTOOLS *Database File Block*, as shown in Fig. 10. The format also includes other attributes of the frames like the transmitting and receiving FlexRay controller, frame size, among others. These details can be obtained by referring to *FrameData.csv* generated in the specification extraction stage. Similarly, for the communication tasks, schedules are generated and stored in a format in which it needs to be entered in SIMTOOLS *Database File Block*. However, SIMTOOLS does not allow importing communication task schedules, and therefore, they are manually entered by the developer according to the generated file. In our opinion, this process can be made completely automated by collaborating with tool suppliers.

Following the parameter write back, the developed software model can be tested using the plant models via closed-loop simulations. SIMTOOLS offers a *Simulation Configuration Block* that enables validation of timing behavior in addition to functional correctness.

3.2.5 Application Software Modeling.

The model developed so far contains details of the controlled plants that will not be in the final software implementation. Moreover, SIMTOOLS *Split and Build* function does not look inside a subsystem block. Therefore, the controller implementations – that lie inside the *FeedbackController* blocks – must be brought to the first level for generating codes and binaries correctly in the next phase. Co-Flex comprises a tool *Dissemble* that automates the above two processes.

Co-Flex: Dissemble – This tool can be invoked from the *Co-Flex Toolbox* block, as shown in Fig. 9, by checking the corresponding checkbox. It takes the development closer to the implementation by (i) deleting the plant models and (ii) expanding subsystem blocks that represent the controller implementations. However, it might be required to reuse the model developed so far partially or completely for a different car with different requirements. Therefore, it makes sense to not modify the model developed so far and, instead, create a new model by copying only the parts that will be in the implementation. *Dissemble* tool automatically copies the controller implementations and the platform configuration block into a new Simulink model while adding new blocks for clock synchronization. This can be achieved by using the MATLAB function *add_block*.

Furthermore, the model so far may not contain the details of sensor and actuator tasks. They depend on the type of sensors and actuators used. For example, in case of an adaptive cruise control application, the sensor can be a camera and the sensor task requires video processing to detect a slowdown of the vehicle in front. Therefore, in this stage, developers need to manually incorporate such details about sensing and actuation into the software model. In future, standard task models can also be added to *Co-Flex: Model* library to further reduce manual interventions. This final software model with every detail of the applications is then used in the next phase for the code generation and the hardware implementation, respectively.

4 A CASE STUDY

We consider a system motivated by software applications in the automotive domain. Due to confidentiality issues, it is difficult to find a case study that represents a real industrial system and obtain all the details including the mathematical model of the plants. Therefore, we use a synthetic case study, consisting of a FlexRay-based ECU network implementing 5 control applications that are typically found in the automotive domain.

Plant models: We study a set of 5 control applications denoted by $C = \{C_1, C_2, C_3, C_4, C_5\}$. C_1 to C_5 represent respectively a DC motor speed control (DCM), a car suspension system (CSS), an electronic wedge brake (EWB), and two variants of cruise control (CC1) and (CC2). In this work, we have studied only linear plant models, however, in real-world scenarios, we might have piecewise-linear or nonlinear plants. It is to be investigated in the future how to extend the co-design technique, discussed in this paper, to such scenarios. The plants[2] under consideration are described as follows:

(C_1) The DC motor speed control application (DCM) has state variables $x = [x_1\ x_2]^T$ representing respectively the rotational speed of the motor shaft and the armature current. The control input u is the motor terminal voltage. This application can, for example, represent a wheel speed control in an electric vehicle. The system matrices for this plant are given as follows:

$$A^c = \begin{bmatrix} -10 & 1 \\ -0.02 & -2 \end{bmatrix}, \quad B^c = \begin{bmatrix} 0 \\ 2 \end{bmatrix}, \quad C^c = \begin{bmatrix} 1 & 0 \end{bmatrix}. \tag{31}$$

For this plant, we consider that the control objective is to have a value of the cost function, given by Eq. (7), lower than 0.7 in a unit step response. Here, we assume that the value of λ is 0.001.

(C_2) The car suspension system (CSS) has the state variables $x = [x_1\ \ x_2\ \ x_3\ \ x_4]$, where x_1 and x_2 represent the position and velocity of the car, and x_3 and x_4 are the position and velocity of the mass of the suspension system. The control input u is the force applied to the body by the suspension system. The system matrices are given as follows:

$$A^c = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -8 & -4 & 8 & 4 \\ 0 & 0 & 0 & 1 \\ 80 & 40 & -160 & -60 \end{bmatrix}, \quad B^c = \begin{bmatrix} 0 \\ 80 \\ 20 \\ -1120 \end{bmatrix}, \quad C^c = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}. \tag{32}$$

The control objective is to have a unit impulse response with a settling time ξ lower than 1 s.

(C_3) We study a simplified version of an electronic wedge brake (EWB). Two state variables $x = [x_1\ \ x_2]$ are the position and the velocity of the braking wedge, respectively. For a simplified DC motor model, the control input u is the force exerted by the motor. The plant model is given by:

$$A^c = \begin{bmatrix} 0 & 1 \\ 8395.1 & 0 \end{bmatrix}, \quad B^c = \begin{bmatrix} 0 \\ 4.0451 \end{bmatrix}, \quad C^c = \begin{bmatrix} 7992 & 0 \end{bmatrix}. \tag{33}$$

The control objective is to have a unit step response with a settling time ξ lower than 0.2 s.

(C_4) CC1 is a simplified version of a cruise control system, i.e., neglecting the dynamics of the powertrain and tires. Here, the state variable x represents the speed of a vehicle and the control input u is the force exerted on the vehicle. The plant model is given by:

$$A^c = -0.05, \quad B^c = 0.001, \quad C^c = 1. \tag{34}$$

The control objective is to have a unit step response with a settling time ξ lower than 0.5 s.

(C_5) CC2 is a more detailed version of a cruise control system. It regulates the vehicle speed in order to follow the driver's command. The state space representation of this system can be

[2] The references are mentioned in [36].

Table 1. Task mapping

E_i	Tasks
E_1	$\tau_{s,1}$, $\tau_{c,2}$, $\tau_{a,3}$, $\tau_{a,4}$, $\tau_{c,5}$.
E_2	$\tau_{a,1}$, $\tau_{s,2}$, $\tau_{c,3}$, $\tau_{s,4}$, $\tau_{s,5}$.
E_3	$\tau_{c,1}$, $\tau_{a,2}$, $\tau_{s,3}$, $\tau_{c,4}$, $\tau_{a,5}$.

Table 2. FlexRay Bus Configuration

Parameters	Values
Bus Speed	10 Mbps
T_{bus}	5 ms
MacroTick	1 µs
N	25
Δ	100 µs

Table 3. Task WCETs

C_i	WCET in µs		
	$\tau_{s,i}$	$\tau_{c,i}$	$\tau_{a,i}$
C_1	200	300	100
C_2	400	600	100
C_3	200	300	100
C_4	100	150	100
C_5	300	450	100

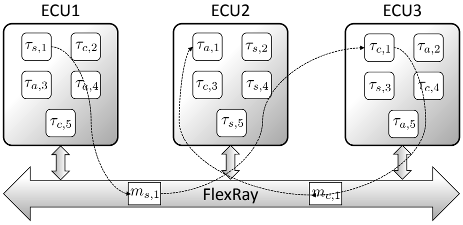


Fig. 11. Platform architecture for the case study.

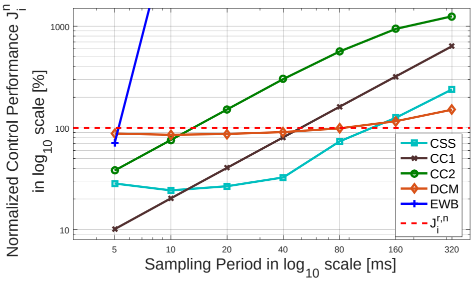


Fig. 12. Prospective Optimal Control Design.

described as follows:

$$A^c = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -6.0476 & -5.2856 & -0.238 \end{bmatrix}, \quad B^c = \begin{bmatrix} 0 \\ 0 \\ 2.4767 \end{bmatrix}, \quad C^c = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}. \tag{35}$$

The control objective is to have a unit step response with a settling time ξ lower than 0.5 s.

Platform architecture: In this case study, we use a hardware platform, as shown in Fig.11, that consists of three ECUs connected by FlexRay in a bus topology. For the experiments, we have considered FlexRay 2.1 because SIMTOOLS V5.2.0, which we have used to develop the software, is not compatible with FlexRay 3.0.1. Table 1 shows the tasks mapped on the ECUs. Table 2 shows the bus parameters that we have taken from a related work [39]. Table 3 shows the assumed values of WCETs for all the tasks. We have further assumed that the WCET of a communication tasks is ϵ = 300 µs. The bus topology is a common one in the automotive domain, as seen in related works [16, 19, 39]. The number of ECUs, the number of control applications, and the task attributes are chosen for the ease of demonstration. The applicability of the proposed methods, however, is not limited to this setup. In [36], a scalability analysis of the co-design scheme is provided where several different setups with varying system sizes and task mappings are considered.

Controller design: Fig. 12 depicts the variation of optimal normalized control performance with sampling period for each control application. Except for EWB, where controllers stabilizing the plant can only be found for a sampling period of 5ms, stable controllers can be designed for each application at all possible sampling periods including 5 ms, 10 ms, 20 ms, 40 ms, 80 ms, 160 ms, and 320 ms. The red dashed line in the plot shows the normalized required performance for all applications (i.e., 100%). Only the points on or below the red line meet the performance requirements and only these points will be considered in the co-optimization stage.

In this stage, for each application and at each possible sampling period, a search of poles for the equivalent discrete-time closed-loop system is carried out. Fig. 13 illustrates the influence of the granularity of a pole search. As we can observe in the figure, the finer the granularity is, the more

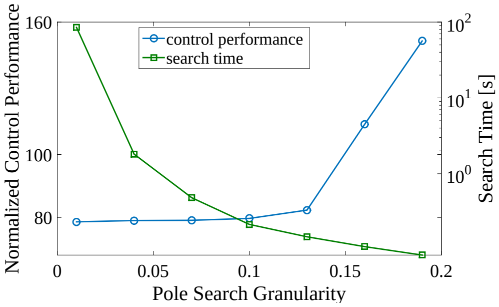


Fig. 13. Granularity of pole search for DCM.

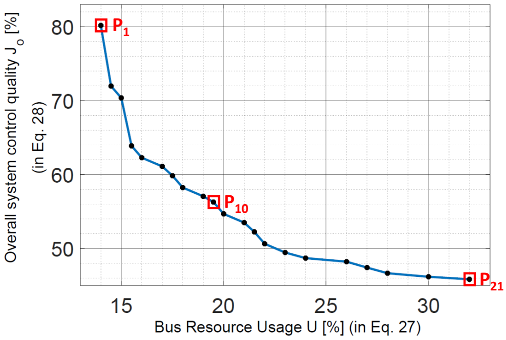


Fig. 14. Pareto front.

time it requires to search the pole space, and controllers with better control performance can be found. The designed controllers, hence, might be sub-optimal. By tuning the control gains, it might be possible to design a better controller at the cost of more manual efforts. It should be noted that the order of a system also influences the search time. With the same granularity, the higher the system order is, the longer it would take to search the pole space. A second-order plant (i.e., DCM) is studied to obtain the results shown in Fig. 13.

Co-optimization: Fig. 14 shows the Pareto front that we have obtained in the co-optimization stage. It can be observed that 21 reasonably well-distributed Pareto points are generated. Each point represents a design option that offers a certain trade-off between the bus resource usage and the overall QoC. The developer can choose a design option according to the requirements. On the other hand, using the conventional approach as discussed in Sec. 2.3, the developer, in the best case, might come up with a design configuration equivalent to P_1 (when optimizing the resource usage) or P_{21} (when optimizing the control performance). Hence, Co-Flex offers more design freedom compared to the conventional approach.

The values of bus resource usage – percentage of static slots used – range from 14% to 32%. The values of overall QoC – average control performance – vary from 45.82% to 80.14%. Note that for the performance measures considered in this paper, the smaller the value is, the better is the performance. If using less resources is the top design priority, the engineer can only use 14% of the bus bandwidth in the static segment to achieve stable control. If a performance-optimal design is desired, a much better performance (average improvement of 45.82%) can be achieved at the cost of an extra 18% of the bandwidth. The Pareto points can be explored to obtain a design that is the most suitable according to the requirements. For a relatively small system size, there is already a considerable design freedom available. For larger systems, developers may have more choices for the trade-off between the design objectives.

Furthermore, it can be observed that for all values of U, we do not get a Pareto point. The reason for this could be that for a value of resource usage, either (i) a feasible parameter set could not be obtained, or (ii) the optimal solution is dominated by other points.

For our case study, the co-optimization stage took 3.56s on a desktop computer with an AMD Athlon (tm) II X2 245e processor of 2.90GHz and 4GB RAM. The optimizer is implemented in Matlab. We use Gurobi [22] in the inner optimization layer and CPLEX [24] in the outer layer. In the outer layer, we need all optimal solutions and CPLEX provides a feature called *solution pool* for that, while Gurobi is more efficient in the inner layer than CPLEX.

Software development: In [36], we presented the same case study as here, where different steps of software development were manually implemented. Here, we use the proposed toolchain to develop the distributed control software in an automated fashion. This basically reduces the development

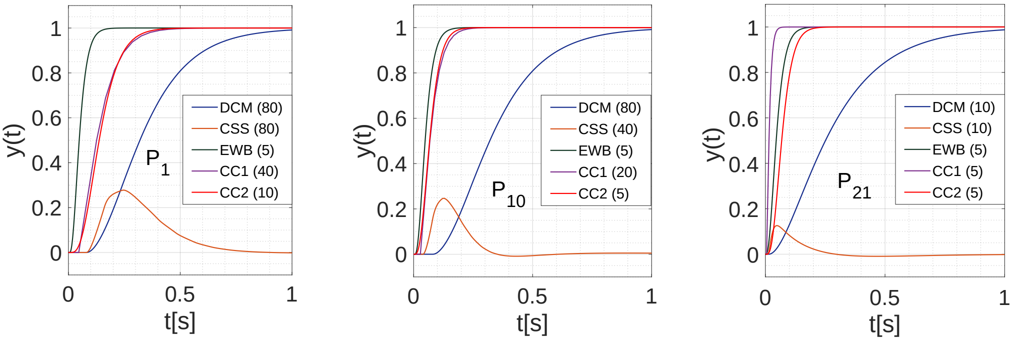


Fig. 15. Control responses using the Pareto points P_1, P_{10} and P_{21} (Sampling periods [in *ms*] are given in brackets).

time from 5 hours to less than 20 minutes for this case study. For industrial-sized systems, the software development time can be reduced from several days to a few hours. Note that the maximum time is invested in modeling the specification (i.e., in parameterizing the template blocks). In the future, we can also automate the development of specification models using information contained in excel sheets.

Based on the methodology proposed in Sec. 3, we have designed and implemented the software for the system under study. We have first developed the specification model including the plant models using SIMTOOLS and Co-Flex in the Simulink environment as discussed in Sec. 3.1. Subsequently, all relevant information is extracted (Sec. 3.2.1) and the proposed co-optimization approach is applied (Sec. 3.2.2 and Sec. 3.2.3). From the obtained Pareto front, we have selected 3 well-spaced Pareto points, i.e., P_1, P_{10} and P_{21} as marked in Fig. 14. The system model – comprising the controller templates and the controlled plants – is configured successively with parameters corresponding to these Pareto points, as described in Sec. 3.2.4, and tested for plausibility. The fully specified models corresponding to the 3 Pareto points are successively simulated according to the 4^{th} level of simulation available in SIMTOOLS, where the ECUs and the communication system are simulated based on the timings of application tasks, communication tasks and bus schedules.

The system responses are recorded for each of the simulation runs and are shown in Fig. 15. Here, the control systems C_1, C_3, C_4, and C_5 are applied unit step references while the system C_2 is applied an unit impulse reference. It may be observed in the figure that in the case of CC2, the system response corresponding to P_1 is different from the responses (identical) corresponding to P_{10} and P_{21} respectively. This is because the synthesized sampling period of CC2 corresponding to P_{10} and P_{21} is the same and is equal to 5 ms while the sampling period value corresponding to P_1 is 10 ms. The responses of CC2 are identical for P_{10} and P_{21} despite different task and message schedules. This verifies our assumption in the co-optimization stage that the control performance depends only on the sampling period of the control application. Furthermore, we can observe that the settling times in the system responses of CC1 are respectively 52 ms, 212 ms, and 413 ms corresponding to the sampling periods of 5 ms, 20 ms, and 40 ms and are approximately in the ratio 1:4:8. This can be also verified from the control performance curve shown in Fig. 12. Similarly, for all applications, control performances are verified against the values obtained in the prospective controller design stage.

After the simulations, we have used the tool *Dissemble* to automatically remove the plant models and expand the subsystem blocks, as described in Sec. 3.2.5. Further, we have generated C-code and binary files successfully using SIMTOOLS *Split and Build*, Simulink RTW, and SIMTARGET. *The binaries files thus obtained are then used to flash three Elektrobit (i.e., EB 6120) ECUs. These ECUs are connected over a FlexRay bus (i.e., using unshielded twisted pair cables and D-SUB9 connectors).* We have not encountered any error, which implies that the parameters have been correctly configured.

5 RELATED WORK

Towards meeting the high performance and low cost requirements for embedded control systems, co-design of control algorithms and their platform implementations – often referred to as a cyber-physical systems design approach – is the key [6, 7, 17, 37, 53]. Such co-design also helps reduce testing and integration efforts when implementing control algorithms. While co-synthesis has already been studied in the general embedded systems context [50], several control/architecture co-design methods have recently been proposed [16, 36, 38]. [38] has proposed a method that integrates controller design with task and message scheduling while optimizing the overall control performance. Later, [16] has put forward a co-design problem formulation for FlexRay-based control systems also with control performance as the only optimization objective. This work considers both sampling period and delay as variables during the controller design. Note that [16, 38] have not considered the trade-off between multiple optimization objectives. Also, some existing approaches appear difficult to scale. For example, in [16], it already takes more than one hour to synthesize a system of 5 applications, amongst which 3 are control applications. In our proposed toolchain, we have implemented the co-design approach from [36]. It scales to industrial-sized systems (e.g., a bus cluster) and co-optimizes overall QoC and resource usage, respectively. Note that none of these works have extended a co-design framework into a full-fledged toolchain support compatible with COTS software development tools.

For this work, we have reviewed existing industrial toolchains for automotive software development. Major Tier 2 automotive suppliers include Vector, dSPACE, Elektrobit, and Siemens Industry Software (formerly known as Mentor Graphics). Vector offers PREEvision [45], where a software architecture and a mapping of software components to ECUs can be specified and an in-vehicle network can be configured. Specification for each ECU can be extracted from PREEvision and imported into DaVinci Configurator Pro [44], where it is integrated with a basic software and the codes for software components generated from model-based design tools like Simulink. Developers need to configure the runtime environment for ECUs in DaVinci Configurator Pro [44] and then binary files can be generated that will be used to flash the ECUs. A similar tool flow is offered by Siemens Industry Software where Capital [30] allows architecture and network design while Volcano Vehicle System Builder [31] enables software integration for ECUs. In the same vein, dSPACE provides SystemDesk [9] for architecture design and TargetLink [10] for functionality development and code generation. In this work, we have studied software development for FlexRay-based distributed control systems using MATLAB/Simulink and SIMTOOLS/ SIMTARGET [4, 41, 42], as discussed in Sec. 2.3. Using the aforementioned toolchains, controllers are designed in model-based design tools while the schedule configuration for ECUs (e.g., in DaVinci Configurator Pro and Volcano Vehicle System Builder) and communication buses (e.g., in PREEvision and Capital) are realized in different tools. Unlike these toolchains, Co-Flex offers to co-design control and platform parameters.

In the context of hardware/software co-design, there also exists academic tools like Metropolis [2], Metro II [8], Ptolemy II [11], and Metronomy [21] that allow modeling of heterogeneous components and their co-simulation, thereby enabling a design space exploration for heterogeneous systems. Metropolis offers a unified language for capturing different models of computation like dataflow, state machine, and discrete time. Later, Metropolis was extended to Metro II that also allows different specification language for different components. For example, in the case of a building design automation, a controller can be specified as a Simulink model, the specification for a physical building can be in Modelica, while an architecture can be modeled in Metro II [49]. The controller and the building models can be co-simulated with the computing platform in Metro II. Both Metropolis and Metro II use a SystemC based engine for the simulation [1]. Ptolemy II can also be used to model functions and architecture in an integrated framework, however, it is more focused on functional modeling allowing several models of computation including process networks, synchronous reactive and continuous time. Metronomy combines the advantages of Ptolemy II and Metro II respectively and allows to model functions in Ptolemy and architectures in Metro II. Besides these tools, TrueTime [5] is a MATLAB/Simulink toolbox that enables to simulate embedded control systems. Also, Blech [20], an imperative synchronous programming language, can be used to develop safety-critical embedded control software that can be verified for different functional and timing properties. None of the aforementioned tools offer to co-design controllers and their platform implementations. Moreover, Metropolis, Metro II, Ptolemy II, and Metronomy are based on the principle of separation of concerns. We believe that the co-design technique presented in [36] can be integrated into these tools. However, the above tools do not have in-built models for the target platform under study and the corresponding basic software. Hence, we have considered a commercial toolchain that can be used to develop the software for a distributed platform comprising EB 6120 ECUs connected over a FlexRay bus.

6 CONCLUDING REMARKS

In this paper, we have introduced a software development process for FlexRay-based distributed control systems. It enables correct-by-construction and convenient design and implementation of such systems. We have also developed an integrated toolchain that automates the development process to a large extent. It integrates a recently proposed control and platform co-design scheme into available COTS development tools for control systems and embedded software. Therefore, it reduces manual effort and improves design optimality.

Our proposed toolchain is only a preliminary one, showing that such an integrated design and implementation of distributed embedded controllers is possible in an automotive setting. We would like to extend the toolchain to address a more comprehensive design problem that includes frame packing, task partitioning and mapping. Moreover, the toolchain, in its current state, only considers LTI feedback control systems. In the future, we can consider nonlinear systems and control techniques like model predictive control (MPC), gain scheduling and adaptive control. For complex controllers, e.g., an MPC, the WCET of a controller might be different for each sampling period. Note that the co-design technique used in Co-Flex can easily accommodate the scenario where a WCET is a function of the sampling period. This is because in the inner layer, the sampling period is known, and therefore, the corresponding WCET can be used for scheduling. In this work, we have considered that the WCET of a task is given, however, an automated toolchain would need a WCET estimator.

In industrial systems, a controller might switch between multiple operational modes depending on the changes in the physical system (e.g., plant dynamics might change based on environmental factors like temperature and wind speed) or in the cyber system (e.g., overloads in ECUs and communication buses). While [3] has considered integrated modeling and verification of such control loops using the theory of hybrid automata, [18, 34, 35] have shown how to optimally dimension communication resources shared by such multi-mode controllers. An optimal co-design of control and platform parameters considering multi-mode controllers can be studied in the future.

While we have studied SIMTOOLS and SIMTARGET for the design of FlexRay-based systems, we may investigate how our proposed development process extends to other communication systems like CAN and time-triggered Ethernet. We may also study a more complex tool flow for software synthesis of distributed control systems on platforms comprising multiple different communication buses and allowing different ECU scheduling policies. We hope that this work motivates further research on closing the gap between the state of the art and the state of practice in designing distributed embedded controllers.

Acknowledgements: This work was supported by: (i) the NSF Award #2038960 "CPS: Medium: GOALI: Design Automation for Automotive Cyber-Physical Systems", (ii) the AvH Foundation through the Chair for Cyber-Physical Systems in Production Engineering at TUM, and partially by (iii) the DFG project SFB 768.

REFERENCES

[1] F. Balarin et al. 2002. Concurrent execution semantics and sequential simulation algorithms for the Metropolis meta-model. In *International Symposium on Hardware/Software Codesign (CODES).*

[2] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. 2003. Metropolis: An integrated electronic system design environment. *Computer* 36, 4 (2003), 45–52.

[3] M. Bitzer, M. Herrmann, and E. Mayer-John. 2020. System Co-Design (SCODE): Methodology for the analysis of hybrid systems - A systematics for complexity reduction of control software in embedded systems. *Automatisierungstechnik* 68, 6 (2020), 488–499.

[4] C. Phagoo ang G. Freiberger and D. Millinger. 2009. System design validation using Matlab/Simulink and EB Simtools at Ford Motor Company. White Paper. https://www.all-electronics.de/wp-content/uploads/migrated/article-pdf/84620/532ag1109.pdf

[5] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. 2003. How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime. *IEEE Control Systems Magazine* 23, 3 (2003), 16 – 30.

[6] S. Chakraborty, M. A. Al Faruque, W. Chang, D. Goswami, M. Wolf, and Q. Zhu. 2016. Automotive Cyber-Physical Systems: A Tutorial Introduction. *IEEE Des. Test* 33, 4 (2016), 92–108.

[7] W. Chang, L. Zhang, D. Roy, and S. Chakraborty. 2017. *Control/architecture codesign for cyber-physical systems.* Springer.

[8] A. Davare et al. 2013. MetroII: A design environment for cyber-physical systems. *Transactions on Embedded Computing Systems* 12, 1s, Article 49 (2013), 31 pages.

[9] dSpace. 2021. SystemDesk: Modeling system architecture and generating virtual ECUs. Online. https://www.dspace.com/en/ltd/home/products/sw/system_architecture_software/systemdesk.cfm#143_25611.

[10] dSpace. 2021. TargetLink: Production code generation for the highest demands. Online. https://www.dspace.com/en/inc/home/products/sw/pcgs/targetli.cfm.

[11] J. Eker et al. 2003. Taming heterogeneity - the Ptolemy approach. *Proc. IEEE* 91, 1 (2003), 127–144.

[12] D. Eller, J. Aggarwal, and H. Banks. 1969. Optimal control of linear time-delay systems. *IEEE Trans. Automat. Control* 14, 6 (1969), 678–687.

[13] M. S. Fadali and A. Visioli. 2009. Chapter 9 - State feedback control. In *Digital Control Engineering*, M. S. Fadali and A. Visioli (Eds.). Academic Press, Boston, 335–378.

[14] FlexRay Consortium. 2010. The FlexRay communications system protocol specification, Version 3.0.1. Retrieved December 23, 2016 from https://svn.ipd.kit.edu/nlrp/public/FlexRay/

[15] M. Gaid, A. Cela, , and Y. Hamam. 2006. Optimal integrated control and scheduling of networked control systems with communication constraints: application to a car suspension system. *IEEE Transactions on Control System Technology* 14, 4 (2006), 776 – 787.

[16] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. 2012. Time-triggered implementations of mixed-criticality automtoive software. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[17] D. Goswami, R. Schneider, and S. Chakraborty. 2011. Co-design of cyber-physical systems via controllers with flexible delay constraints. In *16th Asia South Pacific Design Automation Conference (ASP-DAC).*

[18] D. Goswami, R. Schneider, and S. Chakraborty. 2011. Re-engineering cyber-physical control applications for hybrid communication protocols. In *Design, Automation and Test in Europe (DATE).*

[19] D. Goswami, R. Schneider, and S. Chakraborty. 2014. Relaxing signal delay constraints in distributed embedded controllers. *IEEE Transactions on Control Systems Technology* 22, 6 (2014), 2337 – 2345.

[20] F. Gretz and F.-J. Grosch. 2018. Blech, imperative synchronous programming!. In *Forum on Specification and Design Languages (FDL).*

[21] L. Guo, Q. Zhu, P. Nuzzo, R. Passerone, A. Sangiovanni-Vincentelli, and E. A. Lee. 2014. Metronomy: A function-architecture co-simulation framework for timing verification of cyber-physical systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS).*

[22] Gurobi Optimization. 2014. Gurobi Optimizer V6.0. Retrieved Dec. 23, 2016 from www.gurobi.com

[23] J. Hou and W. Wolf. 1996. Process partitioning for distributed embedded systems. In *International Workshop on Hardware/Software Co-Design (Codes/CASHE).*

[24] IBM. 2014. IBM ILOG CPLEX Optimizer V12.6.2. Retrieved Jan. 16, 2017 from www.ibm.com

[25] M. Lukasiewycz et al. 2013. System architecture and software design for electric vehicles. In *50th Annual Design Automation Conference (DAC).*

[26] M. Lukasiewycz, M. Glaß, P. Milbredt, and J. Teich. 2009. FlexRay schedule optimization of the static segment. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS).*

[27] L. Ma, F. Xia, and Z. Peng. 2008. Integrated design and implementation of embedded control systems with Scilab. *Sensors* 8, 9 (2008), 5501 – 5515.

[28] P. Marti, J. M. Fuertes, G. Fohler, and K. Ramamritham. 2002. Improving quality-of-control using flexible timing constraints: metric and scheduling. In *IEEE Real-Time Systems Symposium (RTSS) 2002.*

[29] P. Marti, R. Villa, J. M. Fuertes, and G. Fohler. 2001. On real-time control tasks schedulability. In *European Control Conference (ECC).*

[30] Mentor, A Siemens Business. 2021. Capital: Enabling the electrical model based enterprise. Online. https://www.mentor.com/products/electrical-design-software/capital/.

[31] Mentor, A Siemens Business. 2021. VSTAR tools. Online. https://www.mentor.com/embedded-software/autosar/tools.

[32] K. J. Åström and R. M. Murray. 2008. *Feedback Systems: An Introduction for Scientists and Engineers.* Princeton University Press, USA.

[33] K. J. Åström and B. Wittenmark. 1997. *Computer-Controlled Systems (3rd Ed.).* Prentice-Hall, Inc., USA.

[34] D. Roy, W. Chang, S. K. Mitter, and S. Chakraborty. 2019. Tighter dimensioning of heterogeneous multi-resource autonomous CPS with control performance guarantees. In *ACM/IEEE Design Automation Conference (DAC).*

[35] D. Roy, S. Ghosh, Q. Zhu, M. Caccamo, and S. Chakraborty. 2020. GoodSpread: Criticality-aware static scheduling of CPS with multi-QoS resources. In *IEEE Real-Time Systems Symposium (RTSS).*

[36] D. Roy, L. Zhang, W. Chang, D. Goswami, and S. Chakraborty. 2016. Multi-objective co-optimization of FlexRay-based distributed control systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS).*

[37] D. Roy, L. Zhang, W. Chang, S. Mitter, and S. Chakraborty. 2017. Semantics-preserving cosynthesis of cyber-physical systems. *Proceeding of the IEEE* 106, 1 (2017), 171 – 200.

[38] S. Samii, A. Cervin, P. Eles, and Z. Peng. 2009. Integrated scheduling and synthesis of control applications on distributed embedded systems. In *Design, Automation & Test in Europe Conference & Exhibition (DATE).*

[39] R. Schneider, D. Goswami, S. Zafar, M. Lukasiewycz, and S. Chakraborty. 2011. Constraint-driven synthesis and tool-support for FlexRay-based automotive control systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS).*

[40] C. Shen and W. Tsai. 1985. A graph matching approach to optimal task assignment in distributed computing systems using a minimax criterion. *IEEE Trans. Comput.* C-34, 3 (1985), 197 – 203.

[41] SIMTOOLS GmbH. 2010. SIMTARGET V5.0.1: C code generation from SIMTOOLS models for CAN, FlexRay, and I/O. Retrieved Oct. 17, 2020 from https://nanopdf.com/download/simtools-blocksets-for-the-matlab-simulink-design-environment_pdf

[42] SIMTOOLS GmbH. 2012. SIMTOOLS V5.2.0: Model-based design tools for FlexRay-based applications. Retrieved Oct. 17, 2020 from https://nanopdf.com/download/simtools-blocksets-for-the-matlab-simulink-design-environment_pdf

[43] T. H. Summers and J. Lygeros. 2014. Optimal sensor and actuator placement in complex dynamical networks. *IFAC Proceedings Volumes* 47, 3 (2014), 3784 – 3789. 19th IFAC World Congress.

[44] Vector. 2021. DaVinci Configurator Pro Version 5.23: Configure, validate and generate AUTOSAR basic software. Online. https://www.vector.com/de/en/products/products-a-z/software/davinci-configurator-pro/.

[45] Vector. 2021. PREEvision Version 10.0: Model-based electric/electronic development. Online. https://www.vector.com/de/en/products/products-a-z/software/preevision/.

[46] B. Vogel-Heuser et al. 2020. BPMN+I to support decision making in innovation management for automated production systems including technological, multi team and organizational aspects. In *IFAC World Congress.*

[47] B. Vogel-Heuser et al. 2020. Interdisciplinary engineering of cyber physical production systems: Highlighting the benefits of a combined interdisciplinary modelling approach on the basis of an industrial case. *Design Science* 6, 5 (2020), 1 – 36.

[48] N. Xue and C. Yuan. 2019. Sensor and actuator placement for large-scale systems: A projection-based formulation. In *American Control Conference (ACC).*

[49] Y. Yang, A. Pinto, A. Sangiovanni-Vincentelli, and Q. Zhu. 2010. A design flow for building automation and control systems. In *Real-Time Systems Symposium (RTSS).*

[50] Licong Zhang, Dip Goswami, Reinhard Schneider, and Samarjit Chakraborty. 2014. Task- and network-level schedule co-synthesis of Ethernet-based time-triggered systems. In *19th Asia and South Pacific Design Automation Conference (ASP-DAC).*

[51] L. Zhang, D. Roy, P. Mundhenk, and S. Chakraborty. 2016. Schedule management framework for cloud-based future automotive software systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA).*

[52] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. 2005. Extensible and scalable time triggered scheduling. In *International Conference on Application of Concurrency to System Design (ACSD).*

[53] Q. Zhu and A. Sangiovanni-Vincentelli. 2018. Codesign methodologies and tools for cyber-physical systems. *Proc. IEEE* 106, 9 (2018), 1484–1500.