TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Realzeit-Computersysteme

Schedule Synthesis for Time-Triggered Automotive Architectures

Florian Richard Sagstetter

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. sc. techn. Andreas Herkersdorf
Prüfer der Dissertation:	1. Univ.-Prof. Dr. sc. Samarjit Chakraborty
	2. Prof. Dr. Zebo Peng, Linköping University, Schweden

Die Dissertation wurde am 14.09.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 23.05.2016 angenommen.

Acknowledgements

This thesis is the result of four years of research at TUM CREATE in Singapore. It would not have been possible without the support of many people.

First, I would like to express my deepest gratitude to my supervisor Prof. Dr. Samarjit Chakraborty for his encouragement, guidance and support during this time. I very much appreciate all the fruitful discussions and his continuous interest in my work. I would also like to thank Prof. Dr. Zebo Peng for taking the time to act as a reviewer for this thesis.

I am also especially indebted to Prof. Dr. Martin Lukasiewycz for his patience when introducing me to research and paper writing from the beginning of my PhD. I am grateful for having had such a dedicated advisor.

I would also like to thank my colleagues from the Embedded Systems group at TUM CREATE for all the pleasant coffee and lunch breaks, and all the technical and non-technical discussions. You made the work environment at TUM CREATE an enjoyable one.

Finally, I would like to thank my girlfriend Florence Chen, my parents Eva and Georg Sagstetter, my brother Michael Sagstetter and my uncle Rudolf Seeberger for supporting and believing in me.

Abstract

Recent years have seen a strong increase in the functionality in vehicles based on software and electronics. A growing number of these functions like advanced driver assistance systems or the upcoming drive-by-wire have strict timing requirements. As a result, a clear trend towards time-triggered systems for safety-critical functionality can be seen in the automotive domain. Time-triggered communication buses such as FlexRay and Automotive Ethernet provide the necessary basis for these systems. Time-triggered systems increase the safety and control quality due to their deterministic behavior and minimal jitter. Moreover, simulation, integration, and testing efforts are significantly reduced due to the predictability of the system. However, determining a time-triggered schedule is a challenging task as various constraints have to be taken into account concurrently.

This thesis investigates various aspects of the schedule synthesis for time-triggered architectures in the automotive domain. It addresses the problems of (1) determining schedules for the automotive FlexRay bus, (2) modular schedule synthesis, addressing the problem of integrating independently developed functionality in a system and (3) concurrent schedule synthesis for various architecture variants. The contributions are as follows.

(1) A schedule synthesis framework for the FlexRay bus is proposed. FlexRay is a latest-generation automotive bus supporting a high bandwidth and a time-triggered communication. The proposed approaches support the recently introduced FlexRay 3.0, while being compatible with the current industry standard version 2.1. Three schedule synthesis approaches are proposed. (a) An Integer Linear Programming approach determining an optimal solution, (b) a heuristic with good scalability, and (c) a schedule integration approach capable of combining previously generated subsystem schedules to a global schedule.

(2) A schedule integration approach exploiting the predictability of time-triggered systems is proposed. It allows to generate individual application or subsystem schedules and integrates them into a global schedule in a second step. As the schedule integration does not change the general structure of the subsystem schedules, the previously defined constraints are not affected. Through this divide-and-conquer approach the complexity is significantly reduced while the system becomes highly composable. This approach allows to add or update applications in an iterative process, measurably reducing the integration efforts. During schedule synthesis, all tasks executed on distributed Electronic Control Units (ECUs) and messages sent on communication buses are taken into account. The proposed framework supports heteroge-

neous architectures based on FlexRay and Automotive Ethernet. Several metrics are presented to optimize subsystem schedules to improve the integration results.

(3) Finally, a multi-schedule synthesis is proposed which allows to generate individual system schedules for different vehicle variants while assigning identical schedules to common parts. It allows to clearly reduce testing and integration efforts as it only has to be done once for common parts. This variant-aware schedule synthesis first identifies commonality between different variants, before applying a schedule synthesis for common parts, and finally, integrating uncommon parts. As the complexity of multi-schedule synthesis is significantly increased compared to conventional approaches, we also propose a divide-and-conquer approach based on schedule integration to partition the problem, thus, clearly improving the scalability.

Kurzfassung

In den letzten Jahren hat die Bedeutung von Software und Elektronik durch die Einführung neuer Funktionalität im Automobil stark zugenommen. Eine steigende Anzahl dieser Funktionen hat strikte Anforderungen an das Echtzeitverhalten des Systems. Hierzugehören etwa komplexe Fahrerassistenzsysteme oder zukünftige *Drive-by-wire* Systeme. Dies hat zu einem klaren Trend zu zeitgesteuerten Systemen für sicherheitskritische Anwendungen im Fahrzeug geführt. Zeitgesteuerte Bussysteme, wie FlexRay und Automotive Ethernet, bilden die Grundlage für die darunterliegende Architektur. Ein zeitgesteuertes System erhöht die Sicherheit und die Qualität von Regelungsalgorithmen durch sein deterministisches Verhalten und minimalen Laufzeitabweichungen. Außerdem ist der Aufwand für Simulationen, die Integration von Komponenten und das Testen durch das determistische Verhalten des Systems geringer. Allerdings ist die Bestimmung eines Ablaufplanes für zeitgesteuerte Systeme eine herausfordernde Aufgabe, da eine Vielzahl an Randbedingungen eingehalten werden müssen.

Die vorliegende Doktorarbeit beschäftigt sich mit verschiedenen Aspekten der Ablaufplansynthese für zeitgesteuerte Systeme im Automobilbereich. Die Arbeit befasst sich mit den folgenden Problemen: (1) Bestimmung von Ablaufplänen für den FlexRay Bus. (2) Untersuchung eines modularen Ansatzes für die Ablaufplansynthese, der sich mit der Integration unabhänging von einander entwickelter Funktionalität in ein gemeinsames System beschäftigt. (3) Untersuchung einer Ablaufplansynthese für verschiedene Varianten eines Systems. Die Arbeit leistet dabei die folgenden Beiträge:

(1) Vorstellung einer Ablaufplansynthese für die Nachrichtenübertragung mit dem FlexRay Bus. FlexRay ermöglicht eine hohe Bandbreite und eine zeitgesteute Kommunikation. Die entwickelten Methoden unterstützen die kürzlich vorgestellte Version 3.0 von FlexRay, während sie gleichzeitig auch für den aktuellen Industriestandard 2.1 anwendbar sind. Drei Methoden mit verschiedenen Eigenschaften werden vorgestellt. (a) Ein Verfahren basierend auf Ganzzahliger linearer Optimierung das eine optimale Lösung findet. (b) Eine Heuristik mit hoher Skalierbarkeit. (c) Ein Verfahren zur Integration von Ablaufplänen, welches erlaubt zuvor erstellte Teilsystemablaufpläne in einen gemeinsamen Ablaufplan zu integrieren.

(2) Vorstellung einer Methodik zur Integration von Ablaufplänen in einen gemeinsamen Ablaufplan. Sie erlaubt die unabhänginge Erstellung von Ablaufplänen für einzelne Anwendungen oder Teilsysteme und integriert diese in einen gemeinsamen globalen Ablaufplan zu einem späteren Zeitpunkt. Die Grundlage für diesen Ansatz bildet das deterministische Verhalten zeit-

gesteuerter Systeme. Da während der Integration die generelle Struktur des Teilsystemablaufplans nicht verändert wird, sind dabei alle zuvor definierten Randbedingungen gültig. Durch die Unterteilung der Ablaufplansynthese in mehrere Schritte wird die Komplexität und damit die Laufzeit reduziert, während das Systemdesign modular wird. Diese Methodik erleichtert es daher, Anwendungen zum System hinzuzufügen oder zu aktualisieren. Das vorgestellte Verfahren berücksichtigt sowohl Tasks auf Steuergeräten als auch Nachrichten, die auf den Kommunikationsbussen ausgetauscht werden. Es unterstützt sowohl FlexRay als auch Automotive Ethernet. Verschiedene Metriken werden vorgestellet, die es erlauben die Ablaufpläne von Teilsystem für die spätere Integration zu optimieren.

(3) Vorstellung einer Methodik zur Erstellung von Ablaufplänen für verschiedenen Varianten eines Systems. Bei der Ablaufplansynthese werden unabhängige Ablaufpläne für jede Variante erstellt, die für gemeinsame Tasks und Nachrichten einen identischen Ablaufplan definieren. Dies verringert den Aufwand für das Testen der Varianten und für die Integration gemeinsamer Komponenten in verschiedene Varianten. Die Methodik ermittelt zunächst Gemeinsamkeiten in verschiedenen Varianten, bevor für diese gemeinsamen Teile ein gemeinsamer Ablaufplan erstellt wird. Anschließend müssen Teile in denen sich die Varianten unterscheiden in den Ablaufplan jeder Variante integriert werden. Da die Komplexität der Ablaufplansynthese dadurch höher als bei konventionellen Ansätzen ist, beinhaltet die Methodik auch eine Partitionierungsheuristik, die es ermöglicht, die Ablaufplansynthese in kleinere Teilprobleme zu partitionieren. Die vorgestellte Integrationsmethodik erlaubt dann die einzelnen Teile in einen gemeinsamen Ablaufplan zu integrieren.

Contents

List of Symbols

Notation

x, $\tilde{x}$,x_y,$\tilde{x}_y$	constant which might have an index y
X	set of constants $x \in X$
$\mathcal{X}$	set of sets $X \in \mathcal{X}$
$\mathbf{x}$	variable
$\bar{\mathbf{x}}$	binary variable
$x(\cdot)$	function returning constant x
$X(\cdot)$	function returning set X
δ_x	interval $[\underline{\delta_x}, \overline{\delta_x}]$ where $\underline{\delta_x}$ denotes the lower and $\overline{\delta_x}$ the upper bound of the interval
Δ_x	set of intervals
$G_x = (Y_x, E_x)$	graph where Y_x represents the set of nodes of type y and E_x the set of edges
$\check{x}$	predecessor of node x in the path of a graph connecting x with other nodes
$\hat{x}$	successor of x in the path of a graph connecting x with other nodes

Symbols

a	application
A_{cl}	set of applications in cluster cl
A_{Cl_λ}	set of applications in set of clusters Cl_λ
A_d	set of applications defined by specification d
A_λ	set of applications defined in the comprehensive task graph λ
b_m	base-cycle of a FlexRay message m
c	cycle index for FlexRay schedule
C_f	set of cycles occupied by frame f
c_{fr}	FlexRay parameter defining the number of communication cycles
cl	cluster representing a subset of applications
Cl_λ	set of clusters of the comprehensive task graph λ
$C_{(sl,\check{r})}$	cycles in slot sl suitable to transmit messages sent by $\check{r}$
D	set of specifications
d	specification
D_{IIS}	subset of clusters defining an Irreducible Inconsistent Set (IIS)
$\mathcal{D}_{IIS}$	set of all Irreducible Inconsistent Sets (ISSs)
$D(p)$	returns all specifications the process p is part of
e	edge
E_a	set of edges (data-dependencies) of application a
E_{cl}	set of edges (shared resources) between the applications in cluster cl
E_{Cl_λ}	set of edges (shared resources) between the applications in set of clusters Cl_λ
E_d	set of edges (data-dependencies) of specification d

E_{IISs}	set of edges (data-dependencies) which are part of the Irreducible Inconsistent Sets (IISs)
E_λ	set of edges (data-dependencies) in comprehensive task graph λ
$\check{E}(\phi, p)$	returns all edges (data-dependencies) connecting p with the source process for the path ϕ
$\hat{E}(\phi, p)$	returns all edges (data-dependencies) connecting p with the sink process for the path ϕ
F	set of frames
f	frame in FlexRay schedule
f_p	finish-time for process p
$\mathbf{f}_p$	variable for finish-time of process p
G_a	graph describing an application a
G_{Cl_λ}	graph describing the set of clusters Cl_λ
G_d	graph describing specification d
G_λ	graph describing an comprehsive task graph λ
h_a	period of application a
h_d	(hyper)-period of specification d
h_{fr}	FlexRay parameter defining the period of a cycle
h_m	period of a FlexRay message m
h_p	period of a process p
$h(p, \tilde{p})$	returns the hyper-period of the two processes p and $\tilde{p}$
$\mathbf{i}_m$	integer variable to indicate to which slot and base-cycle a FlexRay message m is mapped
l_{fr}	FlexRay parameter defining the size of a static slot in bytes
l_m	size of FlexRay message m in bytes
M	set of messages
m	FlexRay message

$M(\check{r})$	returns all messages sent by sender $\check{r}$
n_{all}	theoretical number of static slots in a full FlexRay cycle
n_{fr}	FlexRay parameter defining the number of static slots in a communication cycle
$\mathbf{o}_a$	variable for offset of application a
$\mathbf{o}_d$	variable for offset of cluster d
$O_{(d,\tilde{d})}$	set with feasible values for which the two cluster d and $\tilde{d}$ do not intersect
$\mathbf{o}_m$	variable for offset of FlexRay message m
$\mathbf{o}_p$	variable for offset of process p
p	process (task or message)
P_a	set of processes of application a
P_d	set of processes defined by specification d
P_{IISs}	set of processes which are part of the Irreducible Inconsistent Sets (IISs)
p_{snk}	sink process
p_{src}	source process
P_λ	set of processes in comprehensive task graph λ
$\bar{\mathbf{q}}_{\mathbf{f}_p}$	binary variable indicating if the finish-time $\mathbf{f}_p$ exceeds the process period
$\bar{\mathbf{q}}_{\mathbf{w}_{(p,\tilde{p})}}$	binary variable indicating if the waiting-time $\mathbf{w}_{(p,\tilde{p})}$ exceeds the process period
r	resource
$\check{R}$	set of resources sending FlexRay messages
$\check{r}$	resource sending FlexRay messages
$R(d)$	returns all resources to which processes of specification d are mapped to
R_{eth}	set of all resources that are an Ethernet bus
R_{fr}	set of all resources that are a FlexRay bus

$\check{r}(m)$	returns resource message m is sent from
$r(p)$	returns the resource process p is mapped to
Sl	set of slot indexes
sl	index for FlexRay slot
sl_m	slot message m is transmitted in
s_m	start-time of a FlexRay message m
$\mathbf{s}_m$	variable for start-time of a FlexRay message m
$\mathrm{snk}(\phi)$	returns the sink process for the path ϕ
s_p	start-time of process p
$\mathbf{s}_p$	variable for start-time of process p
$\mathrm{src}(\phi)$	returns the source process for the path ϕ
$\mathbf{t}$	time instant
$\bar{\mathbf{u}}_{(f,sl,b)}$	binary variable indicating if frame f is scheduled in slot sl with base-cycle b
$\bar{\mathbf{v}}_{(\check{r},sl)}$	binary variable indicating ownership of resource $\check{r}$ for slot sl
$\bar{\mathbf{v}}_{(\check{r},sl,c)}$	binary variable indicating ownership of resource $\check{r}$ for slot sl in cycle c
$w_{(p,\tilde{p})}$	waiting-time between the finish-time of a process p and a data-dependent successor $\tilde{p}$
$\mathbf{w}_{(p,\tilde{p})}$	variable for waiting-time between the two processes p and $\tilde{p}$
x_m	x-offset of a FlexRay message within a slot
$\mathbf{x}_m$	integer variable for the x-offset of message m in a FlexRay slot
$\bar{\mathbf{x}}_{(m,\tilde{m})}$	binary variable indicating if message m is placed before or after message $\tilde{m}$ in a slot
$\bar{\mathbf{x}}_{(p,\tilde{p})}$	binary variable indicating if p is started before $\tilde{p}$
$\bar{\mathbf{x}}_{(p,\tilde{p},i,j)}$	binary variable indicating if p is started before $\tilde{p}$ for the indexes i and j

$\bar{\mathbf{y}}_{sl}$	binary variable indicating if slot sl is utilized
$\bar{\mathbf{z}}_{(m,sl,b)}$	binary variable indicating if message m is sent in the slot sl with the base-cycle b
β	multi-schedule
$\Delta_{(d,\tilde{d})}$	intervals for which the two cluster d and $\tilde{d}$ do not intersect
δ_d^{fr}	interval defining feasible cluster offset values for FlexRay communication
$\delta_{(d,m)}^{\text{fr}}$	interval defining feasible cluster offset values for FlexRay message m
$\delta_{\mathbf{o}_p}$	interval defining boundaries for the process offset $\mathbf{o}_p$
$O_{(r,d,\tilde{d})}$	set of intervals for which the two clusters d and $\tilde{d}$ do not intersect on resource r
$\eta(p,\mathbf{t})$	function indicating if a process p utilizes a resource at the time instant $\mathbf{t}$
θ_a	maximum end-to-end delay for application a
$\boldsymbol{\theta}_a$	variable for the end-to-end delay of application a
Λ	set of comprehensive task graphs
λ	comprehensive task graph
Π_d	describes the constraints of a time-triggered schedule for specification d
ρ_f	smallest repetition of messages in frame f
ρ_m	repetition of a FlexRay message
$\sigma(M)$	returns a list containing a sorted message set
$\boldsymbol{\tau}_a$	variable for minimal waiting-time between processes of application a
τ_{fr}	FlexRay parameter defining the temporal duration of a static slot in bytes
τ_m	transmission time of a message m
τ_p	execution-time of a process p

$\tau_{p_{\text{idle}}}$	variable for maximal idle time between processes on all resources
τ_r	variable for lower bound of minimal idle-time between processes on resource r
$\Phi(E_a)$	returns all paths in the task graph of application a
ϕ	path in task graph
$\psi(p)$	returns the routing for a process p transmitted via an Ethernet bus

1 Introduction and Background

Recent years have seen a strong increase in the number of advanced driver assistance functionality in modern vehicles. Functions like adaptive cruise control, lane departure warning, or pre-crash belt pretensioners make vehicles safer and driving more comfortable. These new functions are enabled by a radical change of automotive architectures from merely mechanical systems to strongly electronics and software-based architectures in the last 30 years. Today, innovation in the car industry is largely based on the software implemented on the Electrical/-Electronic (E/E)-architecture of a vehicle.

One of the first advanced driver assistance functions was the Electronic Stability Control (ESC). ESC is a safety function which adapts the behavior of the braking system and the drivetrain if necessary. It integrates the functionality of the Antilock Braking System (ABS), preventing the wheels from locking during braking, and Traction Control System (TCS), preventing the wheels from spinning. Both functions ensure that the wheels keep tractive contact with the road. In addition, ESC as a system improves the stability of a vehicle by preventing oversteer and understeer. This is achieved by braking or accelerating a single wheel to adjust the vehicle dynamics. Figure 1.1 illustrates the E/E-architecture of an ESC system. Several sensors provide data to the ESC controller which calculates the current vehicle dynamics and sends control signals to the braking system[1] or the motor control.

Based on the sensor data from the current vehicle speed (i.e. rotational wheel speed), the steering wheel input, and the brake activation, a control algorithm determines the intended vehicle behavior. A sensor measuring the side angle acceleration allows the controller to estimate the

[1]The braking system today is commonly realized by an electronically controlled hydraulic power unit, but might be replaced by brake-by-wire technology in the near future.

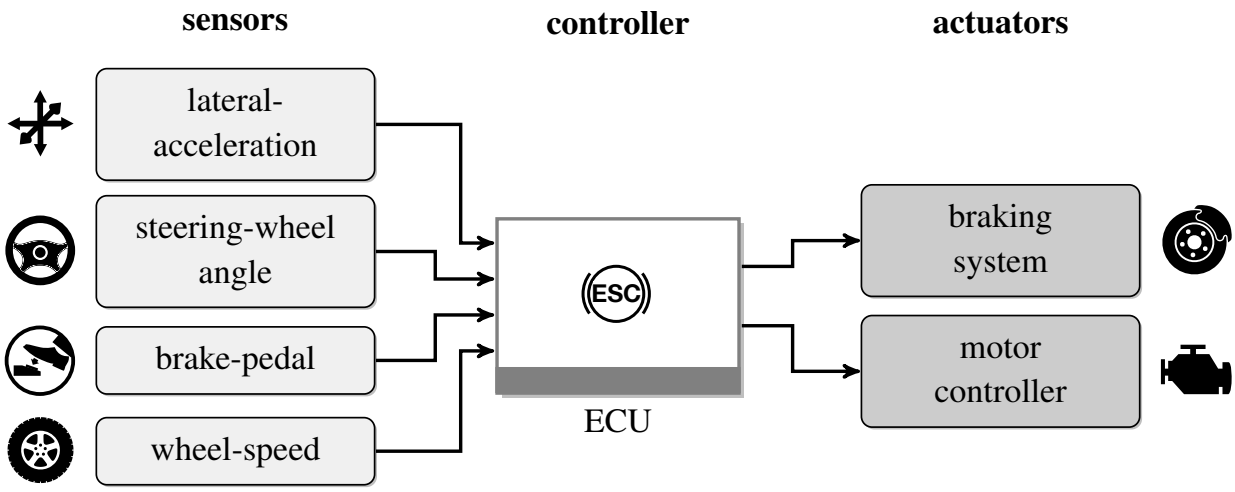


Figure 1.1: *Schematic illustration of the E/E-architecture of an Electronic Stability Control (ESC) system. The ESC controller is connected to multiple sensors and controls the braking system. The ESC controller also provides the motor controller with control instructions and data. The controller is implemented on an Electronic Control Unit (ECU). See [Rob14] for details.*

slip angle deviating from the intended vehicle behavior. Based on this estimate, the controller determines suitable values for the speed of each individual wheel and instructs the brake system to slow down a certain wheel, or the motor controller to accelerate a wheel. ESC is a complex control function which not only has to accurately determine its control instructions but also has to do so in the time frame of a few milliseconds to efficiently stabilize the vehicle [Rob14].

Since the introduction of ESC in 1995, various additional advanced driver assistance functions based on software and electronics have been introduced, like Adaptive Cruise Control (ACC) or pedestrian detection. Today, the electrical and electronic parts of a vehicle already account for more than 30 % of the production costs of a vehicle, despite the decreasing cost of electronic components in general [Cha09, Rob14]. This trend will continue, as most new advanced functions are electronics and software-based. All of these functions have in common that they are highly distributed systems, accessing various sensors and actuators. For instance, for ESC the location of senors and actuators, e.g., the rotational speed sensor at each wheel and the input senor for the steering wheel angle, require a distributed control system. For more advanced functions like ACC, allowing a semi-autonomous driving at low speeds, this effect is even increased, as additional location constraint sensors and actuators, e.g., a radar sensor, are required [BCG12].

Control functions like ESC or ACC are implemented on Electronic Control Units (ECUs). While it would be generally feasible to connect ECUs with point-to-point connections, in reality this is not practical as many functions require common data or interact with each other. Figure 1.2 illustrates the data exchanged between four ECUs: The ESC, the engine control, the

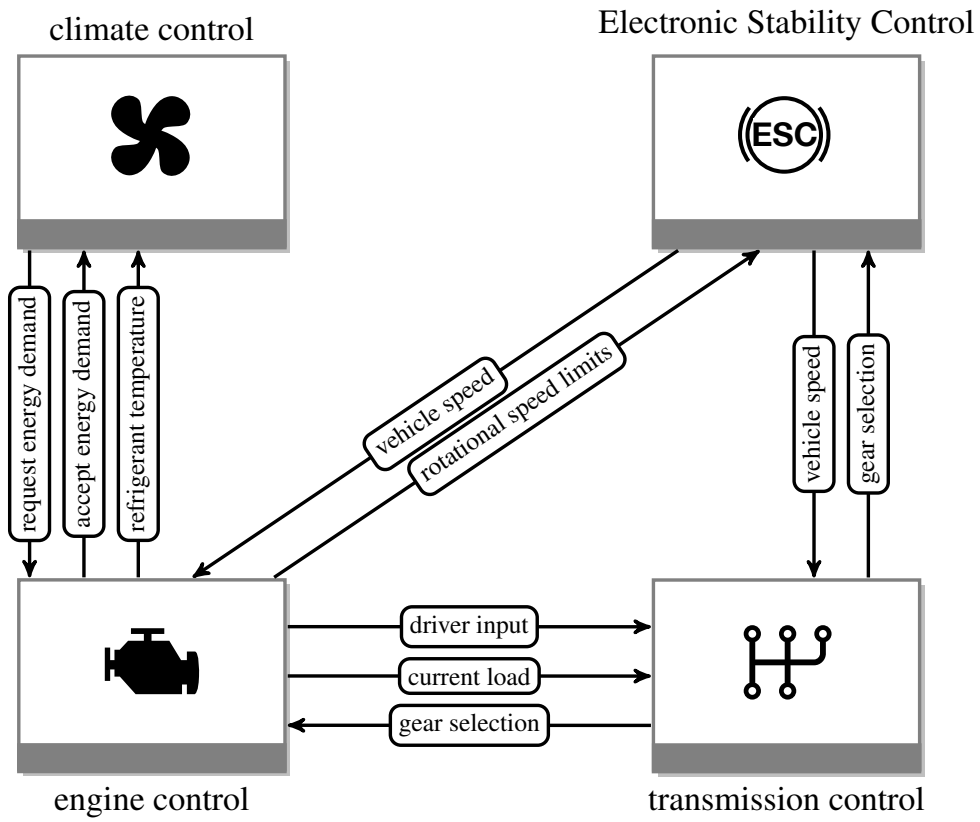


Figure 1.2: *Schematic illustration of data exchanged between* 4 *ECUs. In modern vehicles the communication of an ECU is not limited to the communication with sensors and actuators, but ECUs also exchange various data between each other, both in the form of control instructions as well as information.*

transmission control and the climate control. As mentioned, the ESC controller sends instructions to the engine control, but it also provides the current vehicle speed, calculated from the wheel rotation speed. Hence, instead of having multiple controllers access the wheel rotation speed sensor individually, only the ESC-ECU has direct access to the sensor and provides the calculated vehicle speed to the other ECUs. The ESC also receives the rotational speed limits from the engine controller to ensure that the instructions sent to the engine controller do not stall the engine. Similarly, the engine controller communicates with various ECUs. To fulfill the strict emission requirements and to improve the performance, engine control units of modern vehicles employ highly complex algorithms to determine the optimal point in time and the right amount of fuel to inject [SCTQ09]. The control algorithm therefore not only relies on sensor data of the crankshaft position and the instructions from the gas pedal, but also on the current vehicle speed or the information of the current gear selection in the transmission. For instance, the engine control requires data from the transmission to maintain a minimal engine speed during idle mode, i.e., if no gear is currently selected. But also functionality like the

climate control interacts with the engine control, e.g., to request an increased energy demand, or to adjust the engine cooling [Bor14]. This small excerpt from the in-vehicle communication shows the strong connectivity between different functions. As a consequence, the point-to-point connections between individual ECUs have been replaced by a networked architecture in state-of-the-art vehicles. Communication buses connect multiple ECUs and allow them to exchange messages with each other.

1.1 E/E-architecture

In Figure 1.3, the network structure of an automotive E/E-architecture is illustrated. A state-of-the-art in-vehicle network might consist of up to 100 ECUs connected through shared communication buses [Cha09]. To handle the complexity of more than 2000 individual functions [Bro06], the E/E-architecture is organized in domains. A domain groups functionality based on relations between functions, i.e., data dependencies, as well as safety and hardware requirements, e.g., predictability and reliability. Car manufacturers commonly group their domains in the following four groups: (1) The *comfort* domain containing body and cabin systems, e.g., the climate control. (2) The *infotainment* domain containing display and entertainment systems. (3) The *drivetrain* domain containing drivetrain and emission control systems, e.g., the engine control and transmission control. (4) The *chassis* domain containing vehicle motion, safety and driver assistance functions, e.g., the damper control [Rob14]. A central gateway connects the domains and allows communication between ECUs from different domains[2].

Each domain has very different requirements. For the comfort domain, flexibility and seamless integrability of various individual functions are required [Rob14, ZS14]. The infotainment domain has a high demand in bandwidth to transfer audio or video data [Bor14]. Finally, in the drivetrain and chassis domain the focus is on hard real-time functionality and reliability [Rob14, ZS14]. As a consequence, automotive E/E-architectures are highly heterogeneous systems with a mix of different communication buses, depending on the domain. Latest-generation in-vehicle networks therefore often implement both event-triggered as well as time-triggered systems in the E/E-architecture.

1.1.1 Event-Triggered Systems

An event-triggered system reacts to an incoming event, e.g., the press of a button on the air-conditioning control field, or the motor speed dependent sensor reading for the current piston position of the engine controller [Rob14]. As event-triggered systems are not synchronized with each other, two ECUs might try to access a shared communication bus at the same instant. A collision avoidance in modern communication buses ensures that only one ECU is allowed

[2]Note that the general structure of the network might differ between different car manufacturers, e.g., instead of having a central gateway, domain gateways might be connected by a backbone bus. However, most manufacturers follow a domain-based architecture with a fully connected in-vehicle network. See [Rei14, Rob14] for details.

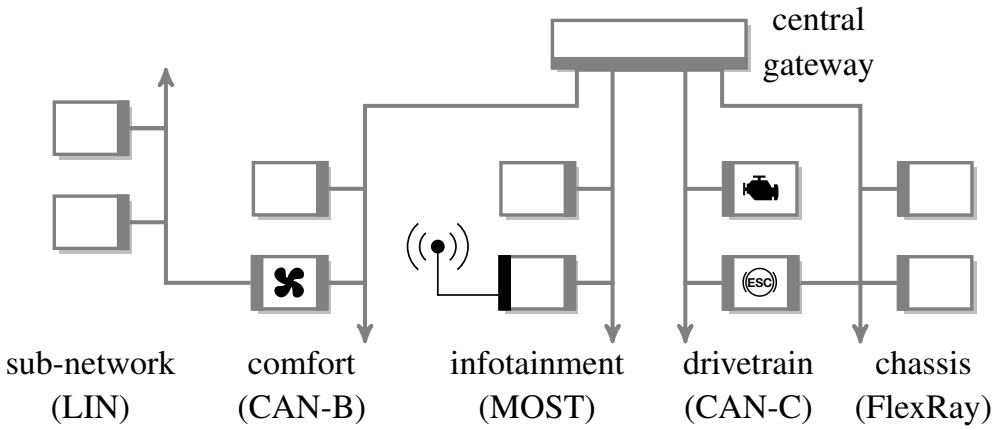


Figure 1.3: *Structure of the E/E-architecture of state-of-the-art vehicles. Sensors and actuators connected to ECUs have been omitted for readability. Modern in-vehicle networks consist of several different bus protocols such as CAN, MOST or FlexRay. See [BMW09] for details.*

to transmit its data and other ECUs have to wait based on predefined priorities. Hence, high priority functionality is processed and transmitted with minimal delays, while functionality with a low priority might have to deal with significant delays if several ECUs with a high priority intend to transmit their data at the same time. Event-triggered systems are well-suited to react to asynchronous (i.e., unpredictable) events. The latency between the occurrence of an event and the reaction is ideally lower than for a time-triggered system. However, depending on the network traffic, the latency can vary strongly, leading to a non-deterministic system behavior [Kop91]. The flexibility of event-triggered systems makes them well-suited for the comfort domain where, depending on the customer's preferences, various different systems with low timing requirements need to be integrated.

1.1.2 Time-Triggered Systems

A time-triggered system is governed by a global time and a schedule which exactly defines at which time instant each function sends its messages or starts software tasks. The deterministic behavior of time-triggered systems is of particular importance for the introduction of electronic *drive-by-wire* systems for steering and braking, replacing hydraulic components. Drive-by-wire systems have strict requirements on the safety, fault-tolerance and availability of the communication system. Time-triggered systems provide these properties. A time-triggered system partitions the time in time-slices during which a function can exclusively utilize a communication bus or an ECU. This provides an isolation between different functionality and guarantees a fixed latency. As messages are transmitted at predefined time instants, the system behavior is predictable and missing messages can be quickly detected. However, time-triggered systems also lead to an increased resource utilization and bandwidth demand, as resources that are

preallocated might not be required at all times. Consequently, communication buses for a time-triggered system should generally support a high bandwidth to ensure that all hard real-time requirements are met [Kop91, Rob14].

1.1.3 Comparison of Event- and Time-Triggered Systems

Extending the discussion of event-triggered and time-triggered systems, in the following a short discussion of the advantages and disadvantages of each is given.

The main difference between event-triggered and time-triggered systems lies in the system scheduler. Event-triggered systems decide at runtime which task or message is released at the current time instant, while time-triggered systems follow a predefined schedule. To determine the system schedule, for time-triggered systems a periodic release is assumed. For the resource requirements pessimistic assumptions, i.e., the Worst Case Execution Time (WCET) for tasks, are made to always ensure the correct functionality of the system. This leads to an overestimation of resource requirements compared to an event-triggered system which only transmits messages when necessary. However, for safety-critical systems timing analysis methods are also applied for event-triggered systems to guarantee the correct system behavior also for worst-case scenarios. These timing analysis methods are also based on worst-case assumptions and a periodic execution. Thus, while event-triggered systems use resources more efficiently than time-triggered systems, to satisfy hard real-time requirements the estimated resource requirements are generally also pessimistic. Consequently, for safety-critical systems, event-triggered systems do not necessarily lead to a higher system utilization [Kop91]. However, as event-triggered systems react to events, for functionality with a high priority in most cases they are able to react faster than a time-triggered system.

As time-triggered systems follow a predefined schedule, the system behavior is predictable and deterministic. Consequently, testing efforts are reduced as the system behavior is easier to reproduce, allowing a systematic testing approach. By contrast, for event-triggered systems a multitude of execution scenarios need to be tested, increasing the testing efforts [Kop91]. The predefined schedule also leads to minimal jitter and allows to design control functions based on a fixed and constant end-to-end delay. Event-triggered systems in contrast require the design of robust control functions which handle a varying end-to-end delay well. In addition, the time-triggered execution scheme of time-triggered systems provides a temporal isolation of functionality. This allows to extend existing schedules if additional messages do not affect previously scheduled ones. While event-triggered systems allow to easily add new functionality, to give timing guarantees as required for critical functions, a timing analysis of the whole system is generally necessary.

In summary, the suitability of event-triggered or time-triggered systems strongly depends on the function requirements. For instance, for non-critical functionality in the comfort domain which is generally highly asynchronous, event-triggered systems are clearly better suited to efficiently use the available resources and react to the user input. For critical applications on the

other hand, the deterministic behavior of time-triggered systems helps to reduce the design and integration efforts.

1.2 Communication Buses

Since the introduction of the Controller Area Network (CAN) bus in a series vehicle in 1991, various automotive communication buses have been introduced in series vehicles. While time-critical functionality in the power train and chassis domains require high bandwidths, e.g., for the ESC to react within a few milliseconds (ms) to a locked wheel, a reaction time of 100 ms is sufficient for the comfort domain, as delays of less than 100 ms cannot be perceived by a passenger. This allows the use of more economic low-speed buses. The infotainment domain on the other hand, requires a high bandwidth of up to 4 Mbit/s for audio data and 50 Mbit/s for video data [Bor14]. As a consequence, the communication protocols used in current vehicles have very different properties. In the following, an overview is given over these bus protocols.

1.2.1 Controller Area Network

The CAN bus is the predominant bus protocol in current vehicles. The CAN protocol is event-triggered and supports messages up to a size of 8 byte, supporting bandwidths of up to 1 MBit/s. A Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) bus access protocol prevents collisions if multiple ECUs try to send their messages at the same time. Predefined priorities define which message is transmitted first in case of a conflict. To ensure a sufficiently short response time of all ECUs, the supported cable length is limited by the bandwidth, i.e., a longer cable supports a lower bandwidth [ZS14]. The CAN bus is used in multiple domains of the vehicle with strongly differing requirements on the network. For instance, in the drivetrain domain a significantly faster data exchange is required compared to the comfort domain where the different ECUs are distributed in the whole vehicle and the cable length of the bus needs to be long [Rob14]. As a consequence, two different CAN versions are currently in use, the *High-speed-CAN (CAN-C)* [CAN13], supporting data rates of 125 kBit/s to 1 MBit/s, and the *Low-speed-CAN (CAN-B)* [CAN06], supporting data rates of 5 kBit/s to 125 kBit/s. Typical applications for CAN-C are the engine control, the transmission control and ESC. Typical applications for CAN-B are the climate control and seat adjustment [Rob14]. The two CAN versions are not compatible.

Due to its limited bandwidth, the CAN bus has become a bottleneck for future functionality such as advanced driver assistance systems. As alternative bus protocols like FlexRay require a partial redesign of existing systems, for legacy reasons, many car manufacturers still rely on the CAN bus [ZS14]. Robert Bosch GmbH has therefore recently proposed CAN with Flexible Data-Rate (CAN FD) which increases the data rate up to 8x, compared to conventional CAN while being backwards compatible [Har12]. CAN FD supports message sizes of up to 64 byte. The bandwidth increase is achieved through increasing the bit rate after the arbitration phase

to transmit the message payload. Hence, legacy ECUs not supporting CAN FD transmit their messages with the bit rate of conventional CAN while CAN FD messages are transmitted at an increased bit rate on the same bus [Bor14, ZS14].

Due to the event-triggered transmission schemes of CAN, the latency of low priority messages can differ strongly. While approaches to determine the maximal latency exist [TBW95, DBBL07], in complex E/E-architectures all constraints are often not known in full detail or might quickly change during the design process [ZS14]. While this makes CAN still suitable for many real-time applications, this non-deterministic behavior makes CAN unsuitable for highly safety-critical applications, like brake-by-wire. As a consequence, Time-Triggered-CAN (TTCAN) was developed, extending the CAN protocol with time-triggered communication [TTC04]. Based on the Time Division Multiple Access (TDMA) scheme, the time is partitioned in time slots. While some time slots are exclusively assigned to an ECU to transmit messages using time-triggered communication, other slots are used for CSMA/CA-based event-triggered communication. A dedicated timing master sends cyclic reference messages to synchronize all ECUs. While TTCAN provides a deterministic communication for critical functionality, it is not compatible with conventional CAN and the low data rate of CAN remains. TTCAN has therefore not found wide acceptance in the automotive industry [Rei14, ZS14].

1.2.2 Local Interconnect Network

The Local Interconnect Network (LIN) bus is a time-triggered low speed bus for simple sensor-actuator-functionality, like door- or seat-electronics [ZS14]. The LIN bus was developed as a low-cost alternative to Low-speed-CAN systems. The goal was to create a simple communication protocol that would not require an additional communication controller for low-end micro-controllers [Rob14]. The communication is triggered by a master ECU, sending cyclic requests to the slave ECUs which respond with the message transmission. The master also triggers the communication between slave ECUs, making the LIN bus a deterministic protocol [Bor14, ZS14]. LIN supports data-rates of $1\,\mathrm{kbit/s}$ to $20\,\mathrm{kbit/s}$. It commonly realizes subsystems, e.g., for climate control, the air-conditioning controller uses the LIN bus to connect to temperature sensors and fans [Rob14].

The latest LIN version 2.2A [LINar] includes various new features such as support for sporadic frames and event-triggered communication, as well as plug-and-play functionality to ease the integration of functionality in different variants [Bor14, ZS14]. While these new features increase the flexibility of the LIN bus, the cost for a LIN controller has also increased and the intended goal to achieve a cost reduction of 50 % compared to a low speed CAN ECU is hard to meet [ZS14].

1.2.3 Media Oriented Systems Transport

The Media Oriented Systems Transport (MOST) bus was specifically designed for multi media applications, providing a high data-rate. For a constant audio and video data transmis-

sion, MOST supports a time-triggered communication [Bor14]. The protocol also supports event-triggered communication for data without a fixed transmission rate, e.g., data of the navigation system [ZS14]. A dedicated master ECU creates the message frames in which the slave ECUs can send their messages. MOST is available in the three variants: *MOST25*, *MOST50* and *MOST150*, supporting data rates of 25 Mbit/s, 50 Mbit/s and 150 Mbit/s, respectively [Grz11]. MOST150 also supports the transmission of Ethernet frames to ease the integration of user devices like mobile phones that already support Ethernet protocols. MOST is currently mainly used in top-of-the-range and mid-range vehicles [ZS14].

1.2.4 FlexRay

FlexRay [Fle13] is a hybrid communication protocol which supports both time- and event-triggered communication. It was developed with highly safety-critical drive-by-wire functionality in mind [ZS14]. FlexRay therefore combines a deterministic communication using a time-triggered protocol with fault-detection and reliability functionality. To provide a redundant message transmission, FlexRay supports two physical channels. In addition, optional *bus guardians* provide a monitoring unit in the ECU to deactivate faulty parts of the network if required [Rei14]. FlexRay provides data rates of 10 Mbit/s for each channel, clearly increasing the bandwidth compared to CAN networks. Like TTCAN, FlexRay uses TDMA for time-triggered communication. For event-triggered communication, a Flexible Time Division Multiple Access (FTDMA) scheme is used. Compared to CAN, the supported network topologies as well as the number of ECUs and cable length is increased through the support of active elements, i.e., a transceiver or repeater [ZS14]. With these properties, FlexRay is suitable for both the drivetrain as well as the chassis domain [Rob14]. For state-of-the-art vehicles, FlexRay is largely used in the chassis domain, for instance see [BMW09].

Despite these advantages, FlexRay is only gradually introduced in series vehicles. The main reasons are the late availability of FlexRay hardware and the slow introduction of drive-by-wire functionality, for which FlexRay is seen as a key enabler. Furthermore, a FlexRay bus configuration requires a set of partially dependent parameters, preventing an easy adoption [ZS14]. A more formal introduction to the FlexRay protocol is given in Section 2.3.2.

1.2.5 Automotive Ethernet

While Ethernet has long been the predominant communication protocol for interconnecting computers, the lack of an implementation fulfilling automotive requirements, such as electromagnetic compatibility, made Ethernet an unsuitable candidate for automotive networks in the past. With the introduction of *BroadR-Reach* [Ope15], an unshielded twisted-pair implementation of the physical layer of Ethernet, these obstacles have been overcome [KCBQ14, ZS14]. As a consequence, all major car manufacturers are currently pursuing the integration of Automotive Ethernet in their in-vehicle networks [Ope15]. However, standard Ethernet does not provide any timing guarantees and is therefore unsuitable for real-time functionality. While several real-time

extensions to standard Ethernet exist, such as PROFINET [PRO] or TTEthernet [TTA08], no mutual standard was defined so far [ZS14]. Therefore, the Time Sensitive Networking (TSN) standards [IEE15], currently under development, define a real-time Ethernet which can also be deployed in the automotive domain. TSN will support time-triggered and event-triggered communication, providing end-to-end performance guarantees, while still supporting standard Ethernet best-effort message transmission [Ste14, IEE15]. Automotive Ethernet currently supports a bandwidth of 100 Mbit/s while a 1 Gbit/s version is under development. The communication is full-duplex, i.e., an ECU can send and receive messages at the same time. Instead of connecting ECUs directly to the Ethernet bus, a central switch is required, allowing the simultaneous communication of independent ECUs. Looking at the full network, Ethernet therefore supports much higher throughputs than the specified base rate [KCBQ14, ZS14].

Ethernet as a communication bus requires a very different design approach compared to existing automotive bus systems. For instance, Ethernet only supports point-to-point connections. Thus, messages between two ECUs are not visible to other network participants. In addition, to support full-duplex, BroadR-Reach employs sophisticated digital signal processing techniques, such as echo cancellation. This makes measuring and analyzing of bus data, as it is done with existing automotive bus systems, not feasible. Concepts for established analysis and diagnosis tools for automotive buses can therefore not be directly applied. Furthermore, the need for switches also increases the cost of an Ethernet network, as switch hardware and additional cabling are required [KCBQ14]. While the introduction of Ethernet might require a radical change in the design of automotive architectures, Ethernet also brings various advantages. As Automotive Ethernet makes a large number of established protocols such as the Internet Protocol (IP) available in the vehicle, the integration of internet and user device related functionality is strongly facilitated. At the same time, the switched network structure makes Ethernet well-suited as a backbone bus, changing the structure of automotive architectures. Finally, the high bandwidths supported by Ethernet allow the introduction of new driver assistance functions, such as the *360-degree camera parking assist system* [ABI14] based on Automotive Ethernet which was introduced by BMW in 2013 [KCBQ14, ZS14]. Analysts are projecting that 40 % of all new vehicles will ship with Ethernet by 2020 [ABI14]. A more formal introduction to Automotive Ethernet is given in Section 2.3.3.

1.2.6 Comparison of Automotive Buses

Extending the introduction of automotive buses in the previous subsections, in the following, a short discussion is given, comparing the different buses. Table 1.1 gives an overview of the different properties of each protocol.

The bus systems deployed in vehicles today have been designed for specific purposes and none of the current bus systems is suitable to replace all other bus types. For instance, the LIN bus is well-suited for subsystems with simple ECUs and replacing LIN with a bus like Ethernet or FlexRay would not be reasonable for practical and economical reasons. Similarly, the CAN

Table 1.1: *Overview of automotive communication buses. See [KCBQ14, Rob14, ZS14] for details.*

	CAN-C High-speed-CAN	**CAN-B** Low-speed-CAN	**LIN**
protocol type	event-triggered[a]	event-triggered	time-triggered
bit rate	$\leq 1\,\text{Mbit/s}$ ($\leq 4\,\text{Mbit/s}$[b])	$\leq 125\,\text{kbit/s}$	$\leq 20\,\text{kbit/s}$
message size	$\leq 8\,\text{byte}$ ($\leq 64\,\text{byte}$[b])	$\leq 8\,\text{byte}$	$\leq 8\,\text{byte}$
number of nodes	10	24	16
network length	40 m	320 m	40 m
topologies	bus	bus	bus
safety-critical	yes	no	no
domain	drivetrain	comfort	comfort
cost	low	low	very low

[a] for TTCAN also time-triggered

[b] for upcoming CAN FD

	MOST	**FlexRay**	**Automotive Ethernet**
protocol type	time- and event-triggered	time- and event-triggered	time- and event-triggered
bit rate	$\leq 150\,\text{Mbit/s}$	$\leq 10\,\text{Mbit/s}$[c]	$100\,\text{Mbit/s}$[d]
message size	$\leq 372\,\text{byte}$/ $\leq 1524\,\text{byte}$[e]	$\leq 256\,\text{byte}$	42-1500 byte
number of ECUs	64	22[f]	-[g]
network length	20 m between nodes	24 m[f]	15 m per link
topologies	ring, star	bus, star, hybrid	star, tree
safety-critical	no	yes	yes
domain	infotainment	chassis	various
cost	high	low	high

[c] for single bus, $\leq 20\,\text{Mbit/s}$ for dual channel, no redundnacy

[d] per link, full-duplex, 1 Gbit/s in development

[e] time-triggered/event-triggered

[f] for passive bus, with active elements up to 2048 ECUs and longer cable

[g] only limited by the number of ports on the switch

bus employed in the comfort domain provides a flexibility and clear cost advantage that is not matched by the other bus systems, making a replacement in the near future not reasonable. On the other hand, in the infotainment domain it is very likely that Ethernet will replace the MOST bus in the near future, as it supports all MOST functionality at similar cost while being also suitable for other domains. Finally, in the safety-critical drivetrain and chassis domains, developers have the choice between CAN FD, FlexRay, and Ethernet. While CAN will still persist as an important part of the automotive in-vehicle networks due to legacy reasons, the introduction of new functionality will require the use of other protocols. For drive-by-wire applications, FlexRay is well-suited and the time-triggered communication supported by TSN might also make Ethernet a suitable candidate for this class of applications. The introduction of advanced driver assistance functions also leads to a strong demand for an increased bandwidth, e.g., to transmit high resolution video data. Currently only Automotive Ethernet is able to provide the required bandwidths if the application is safety-critical. Consequently, we believe that in the near future an increasing number of in-vehicle networks will be using FlexRay and Ethernet. In general, a clear trend towards time-triggered communication for safety-critical systems is visible, while a mix of time- and event-triggered communication will prevail in future automotive E/E-architectures.

1.3 Electronic Control Units

The previous section has discussed the different communication buses in state-of-the-art vehicles, connecting the different ECUs. In the following, the hardware and software of ECUs are introduced in more detail. The ECUs deployed in state-of-the-art automobiles are based on a micro-controller and interface circuits to connect to peripheral sensors and actuators. Depending on the sensors, a preprocessing, e.g., to reduce noise, or a transformation might be required, while an actuator might require power electronics to be controlled [Bor14, Rob14]. Depending on the functionality, the hardware properties of the micro-controllers in use in state-of-the-art vehicles strongly differ. For example, a low-end ECU might only have a 8 bit processor with 8 MHz, 4 kB flash memory, and 256 byte RAM while a top-end automotive ECU might have a 32 bit multi-core processor with 200 MHz, 6 MB flash memory, and 384 kB RAM[3]. An important part of an ECU is the monitoring unit which observes the system behavior and reacts to a fault. In the simplest cases, the monitoring unit might be a watchdog, initializing a reboot of the ECU if required [Bor14]. A state-of-the-art in-vehicle network consists of up to 100 ECUs depending on the car model [Cha09] with very different requirements and functionality.

Instead of implementing the software functionality directly on the micro-controller, ECUs today employ real-time operating systems to manage system resources and to allow the implementation of multiple independent functions on one ECU. For instance, a combustion engine

[3]Data for FREESCALE automotive micro-controllers *S08QD* and *MPC5676R*, taken from http://cache.freescale.com/files/microcontrollers/doc/roadmap/BRAUTOPRDCTMAP.pdf.

controller implements various independent functions, such as the adjustment of the fuel injection while monitoring the rotational speed of the turbo charger [ZS14].

1.3.1 OSEK/VDX

The Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle Distributed Executive (OSEK/VDX) standard [OSE06] defines a Real-Time Operating System (RTOS) which has become the de facto standard for operating systems deployed on automotive ECUs [Bor14]. OSEK/VDX RTOSs are deployed in a wide range of ECUs, ranging from 8 bit to 32 bit processors, however, they are limited to single-core ECUs [Bor14]. OSEK/VDX was designed for distributed architectures, abstracting the communication between both internal and external tasks by messages transmitted using a communication layer. While the standard OSEK/VDX operating system is limited to event-triggered scheduling, a time-triggered version, *OSEK time* is available. There, a predefined *dispatch table* defines at which time instants a task is executed [Rei14]. An OSEK/VDX operating system is configured statically at design time using a configuration file. Dedicated configuration tools commonly allow to auto-generate the configured system [ZS14]. Various real-time operating systems that follow the OSEK/VDX standard are available, for instance see [ETAb, Vec]. However, most OSEK/VDX implementations are incompatible with each other, making a direct migration of software from one system to another a time-consuming task. This is largely due to the fact that, in particular, early versions of the OSEK/VDX specification did not specify all operating system parts in sufficient detail, leading to differing RTOS properties [Bor14, ZS14].

1.3.2 AUTOSAR

To address the increasing complexity of automotive E/E-architectures and to account for the rising number of software-based functionality in vehicles, the AUTomotive Open System ARchitecture (AUTOSAR) initiative was established in 2003. AUTOSAR defines a set of standards [AUT14] for an automotive software architecture which abstracts the underlying ECU hardware to provide a uniform software platform for in-vehicle networks. The goal is to allow developing application software independent of the ECU it is later deployed on. Consequently, functionality could be easily added or updated [Lee09, Rob14].

The AUTOSAR architecture consists of three layers: (1) The hardware specific Basic Software (BSW) layer abstracts the ECU hardware, providing hardware drivers and an RTOS. (2) In the Application Software (ASW) layer the actual system functionality is implemented, e.g., the control algorithm for an ESC system. (3) The VFB layer abstracts the internal, as well as the external communication between the Software Components (SWCs) implemented in the ASW layer [Rei11]. Figure 1.4 illustrates the AUTOSAR architecture. The underlying BSW layer is based on the OSEK/VDX standards [ZDGSV11] and implemented on the ECU by the supplier. The VFB layer is created statically during the design phase and abstracts the communication within an ECU as well as between ECUs connected by a bus. The communication paths are

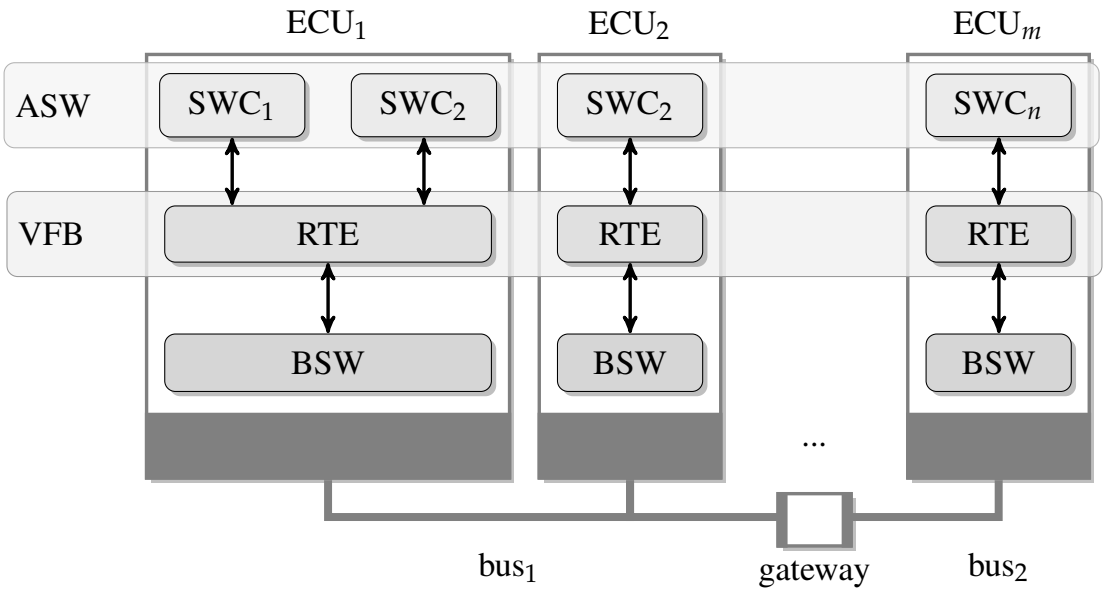


Figure 1.4: *Schematic illustration of AUTOSAR architecture. Each ECU runs a standardized RTOS providing the Basic Software (BSW) layer. The Virtual Function Bus (VFB) layer then abstracts the network structure with its different buses, taking care of all communication. The VFB is implemented by a Real-Time Environment (RTE) deployed on each ECU. Finally, in the Application Software (ASW) layer the actual functionality is implemented as Software Components (SWCs). The figure is reproduced from [Rei11].*

defined in a configuration file from which tools can generate a Real-Time Environment (RTE) deployed on the ECU [ZS14]. For the application development, communication between SWCs is realized using *ports*, both for local as well as external communication. Based on these ports, the Real-Time Environment (RTE) then takes care of the actual data transmission [Rei14, ZS14]. Configuring the VFB is currently still a time consuming task as available configuration tools are unable to auto-generate most of the configuration parameters [ZS14]. AUTOSAR is supported by all major car manufacturers and first ECUs implementing AUTOSAR standards have been introduced into series vehicles since 2009 [Rei14].

1.4 Design of Automotive E/E-architectures

Car manufacturers traditionally rely on suppliers to develop the individual components of a vehicle, such as brakes or seats. The car manufacturer focuses on competitive features such as the vehicle design and the development of the vehicle engine. For the E/E-architecture, most functionality is developed by suppliers and the car manufacturer is mainly concerned with the integration into the system [Lee09, SZ13].

While suppliers today commonly provide ECUs with pre-installed functionality, through AUTOSAR the suppliers might soon provide the functionality as software [ZS14]. The integration of different components in the E/E-architecture is a highly complex task, as a multitude of parameters need to be configured for each ECU and the correct timing behavior of the system needs to be guaranteed. Complex ECUs might have up to 10000 parameters of which several might have to be evaluated experimentally. While tools provide support in the parameter definition, several parameter decisions have to be made by the system designer, e.g., to optimally configure an engine controller [Bor14]. At an early design stage, timing analysis tools such as Symbolic Timing Analysis for Systems (SymTA/S) [Sym15] or methods using Real-Time Calculus (RTC) [TCN00] are commonly employed to determine maximum end-to-end delays. The timing analysis allows to evaluate the specified architecture and to determine conflicts [ST12, ZS14]. During the whole design process, each component, as well as the overall system are then tested to detect problems as early as possible, e.g., by simulating other ECUs in the system, using a residual bus simulation. Tools such as CANoe [Vec15] aid during this process [ZS14]. In recent years, model-based development tools such as Simulink [Mat15] or ASCET [ETAa] are used by most car manufacturers to evaluate their systems at an early design stage. Several of these tools also provide automated C-code generation. As the code generated by these tools currently suffers from being hard to read and being inefficient, it is rarely used in series vehicles [Bor14]. Despite the availability of various tools supporting the development process of the E/E-architecture, the integration of all components and testing of the system is still a time consuming and complex task. Systematic approaches are therefore required to aid this process [SN13, WLMC13].

1.5 Challenges

The number of electronics and software-based functionality in the automotive domain has been rapidly growing in the last years. Figure 1.5 illustrates how the number of ECUs and the number of transmitted signals has significantly increased for the Mercedes-Benz E-Class in the last decades. This trend will continue in the future, and in particular the introduction of advanced driver assistance functions based on video data will lead to a significant increase in bandwidth requirements [ZS14]. This growing number of functionality, but also the increasing interaction between highly distributed software components makes the integration of all components into a seamless working system a challenging task. In this context, in particular three challenges should be mentioned here: (1) Ensuring correct timing behavior of all components individually and in the interaction with other components. (2) The integration of independently developed components in a global system. (3) Efficient management of different variants of an E/E-architecture, e.g., a variant for a petrol and a diesel engine.

The recently introduced ISO 26262 standard [ISO12] for functional safety in the automotive domain requires *freedom of interference*. Hence, during the system design it has to be ensured that the correct execution of a critical software component can never be compromised by another

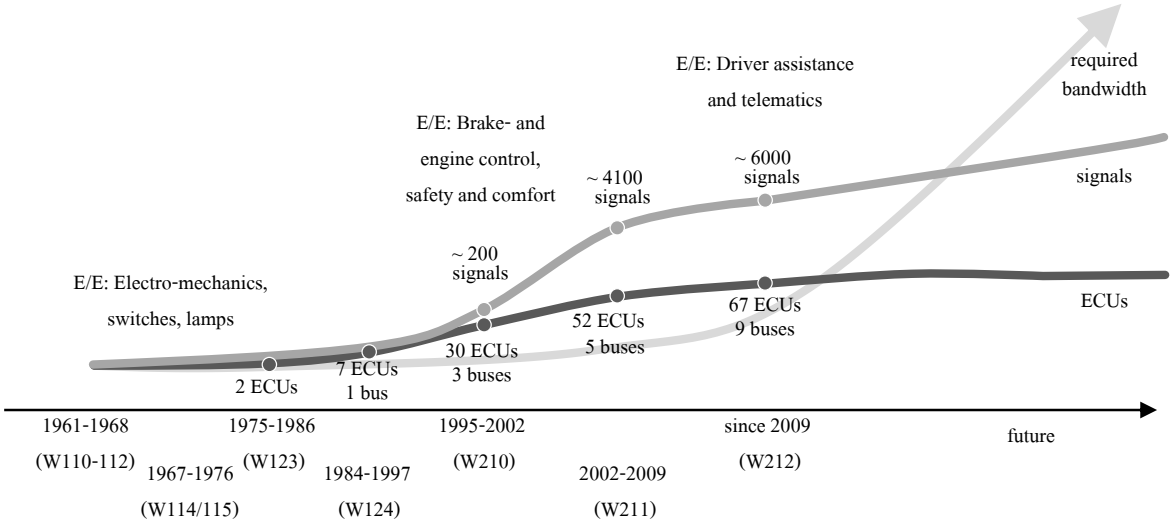


Figure 1.5: *Evolution of the E/E-architecture of the Mercedes-Benz E-Class over the last decades. Figure reproduced from [ST12].*

software component. Consequently, defining and guaranteeing a certain timing behavior becomes essential [SZ13]. AUTOSAR provides mechanisms such as *End-to-End-Communication Protection* to detect timing violations and to contain faults. However, these mechanisms require that the timing behavior of critical software components is clearly defined, making the timing analysis an important part of the software development process [SZ13, ZS14]. Consequently, the design process in the automotive industry is currently radically changing with time and timing behavior becoming an important design parameter. Single software components can no longer be regarded as independent modules that are later integrated in the system. Instead, starting from the specification of the system, the correct functionality in the interaction with other components needs to be evaluated during the whole design process to ensure a sufficient isolation between components.

In the last years, automotive E/E-architectures have seen a strong increase in the number of ECUs. However, this trend will not continue and it is expected that the number of ECUs will decrease from currently around 20-80 ECUs to only 10-20 powerful ECUs. At the same time, the overall number of software functions will continue to grow [Rob14]. As most functionality will then be provided as software components sharing few ECUs, the complexity of the integration process will continue to rapidly increase. The current approach is to apply timing analysis methods for event-triggered systems and define schedules for time-triggered systems already at the specification phase [BÖ07, ZS14]. While this is well-suited for legacy functionality, estimating the exact timing requirements or the communication demand of a software component before the implementation is challenging and might lead to an over- or underestimation of timing requirements. Consequently, during the design process the initial configurations and schedules have to be revised. Tools and methods that allow to efficiently integrate software components

into the global system schedule are essential for the design of future E/E-architectures. While AUTOSAR simplifies the integration process of the software components into a common ECU, configuring the system and defining schedules is still a complex task and systematic methods for the integration are required.

Finally, car manufacturers today offer a large variety of configuration options for their customers, such as different engine variants. For the E/E-architecture of a vehicle model this means that the number of ECUs differs depending on the customer selection, and that for each ECU various configurations need to be maintained. This leads to an almost infinite number of possible configurations [SZ13, Bor14]. To ensure that all these configurations work correctly requires sophisticated testing strategies. At the same time, managing and maintaining all these variants is highly challenging. Consequently, approaches for an efficient variant management taking both commonality as well as variant specific properties into account are required to minimize the testing and integration efforts during the system design.

1.6 Thesis Contributions

In the automotive industry, a clear trend towards time-triggered systems for safety-critical functions can be seen. Time-triggered systems provide a temporal isolation of components [Obe11], making them well-suited to fulfill the requirements of ISO 26262. Consequently, there is an increasing demand for design methods and tools supporting time-triggered systems. While analyzing the timing behavior of time-triggered systems is generally straight-forward due to their predictability, determining a time-triggered schedule for distributed systems is a challenging task. This thesis investigates scheduling methods for time-triggered systems in state-of-the-art and future automotive E/E-architectures. Addressing the problems discussed in the previous section, this thesis makes the following main contributions:

- A schedule synthesis method for message scheduling on the FlexRay bus is presented. The FlexRay bus was designed for upcoming drive-by-wire applications and will therefore play an important role in the next years. FlexRay offers a static segment for time-triggered and a dynamic segment for event-triggered communication. As the dynamic segment is commonly only used for maintenance and diagnosis data, the focus lies on schedule synthesis for the static FlexRay segment. We follow the AUTOSAR specification for FlexRay scheduling, i.e., support a multiplexing scheme. The proposed approaches are among the first supporting all new features of version 3.0 of FlexRay while supporting the still predominantly used FlexRay 2.1, making them backwards compatible.

  Three schedule synthesis methods are proposed with very different properties. (1) A single-stage Integer Linear Programming (ILP) approach that determines an optimal solution but does not scale. (2) A multi-stage ILP for combining previously generated subsystem schedules to a global schedule. It clearly improves the scalability but is not optimal. The multi-stage approach allows to integrate and convert existing FlexRay 2.1

schedules into a FlexRay 3.0 schedule which is important for legacy reasons, i.e., to reduce testing and certification efforts. (3) A greedy heuristic which scales well and obtains high quality solutions in comparison with the optimal solution, but is unsuitable to integrate existing schedules.

A comparison with metaheuristic approaches based on Genetic Algorithms (GAs) or Simulated Annealing (SA) shows the benefits of the proposed approaches. Furthermore, the results show that the new features of FlexRay 3.0 allow to improve the bandwidth utilization.

Finally, we also present an extension of the single-stage ILP for a holistic scheduling. The schedule synthesis then also takes the tasks on the ECUs into account when generating a system schedule.

- Functionality in automotive E/E-architectures today is largely realized through software components distributed over the in-vehicle network. As these functions are developed independently, all components finally need to be integrated into a seamless working system. This is a highly challenging task since all bus and processor constraints as well as end-to-end timing constraints have to be taken into account concurrently. As a remedy, the work at hand proposes a *schedule integration* framework for time-triggered systems. It allows to generate subsystem schedules individually and integrate them into the global schedule at a later stage.

  We present a framework which efficiently realizes the schedule integration. It is based on a Satisfiability Modulo Theories (SMT) approach which integrates the subsystem schedules into a global schedule. As an integration might not always be feasible, we also propose a conflict refinement that identifies conflicting subsystem schedules. The conflict refinement then adapts the subsystem schedule, taking all timing constraints into account to resolve the conflict. The framework is applicable to both FlexRay and Automotive Ethernet. As the ease of integration depends on the structure of the subsystem schedules, we also propose several metrics to optimize subsystem schedules for schedule integration.

  The results of an extensive analysis give evidence of the benefits of the proposed approach compared to conventional approaches, generating a schedule from the scratch. Finally, we also propose a design flow describing how schedule integration might be applied in an AUTOSAR-based development process.

- Car manufacturers provide a growing variety of models and configuration options for customers. Managing these variants and increasing the reuse of functionality in different variants has therefore become one of the key challenges. The work at hand addresses the problem of generating variant schedules for time-triggered E/E-architectures, considering tasks and messages concurrently.

  We propose a *multi-schedule* synthesis approach that determines the common parts of multiple variants and generates a schedule that exploits this commonality. Hence, a

multi-schedule defines individual variant schedules with an identical schedule for tasks common to different variants. This avoids the problem of treating each variant separately, thus, reducing the testing and integration efforts, as it only has to be done once. Multi-schedule synthesis involves several challenges, viz., identification of commonality between different variants, schedule synthesis for common parts, and the integration of uncommon parts. Consequently, the schedule synthesis approach presented here is very different from conventional approaches. The framework relies on a graph-based approach to identify commonality between variants and applies an SMT-based approach for the schedule synthesis. Finally, to address the increased complexity, we also propose a divide-and-conquer approach to partition the problem, improving the scalability. The problem partitioning relies on the schedule integration proposed in this thesis.

The results give evidence of the benefits of the proposed multi-schedule synthesis and the efficiency of the approach. The framework is also evaluated using the lab setup of an automotive system.

1.7 Related Work

This section gives a brief overview of literature related to the schedule synthesis problems addressed in this thesis. A general overview of work related to each of the three specific problems (as described in the previous section) is given. A more detailed discussion of work specifically relevant for the proposed approaches is then given in the respective chapters.

For the design of embedded systems there are two general approaches to handle real-time applications: (1) Following an event-triggered scheme which executes tasks and transmits messages once certain events occur, or (2) to apply a time-triggered scheme which starts tasks and sends messages at predefined points in time. There has been a long debate in the scientific community about the advantages of each approach [Kop91, ATB93, Kop95]. However, the choice of the right approach strongly depends on the application [Alb04, TSPS12]. As a consequence, the last years have seen the introduction of various bus protocols in the automotive domain which support both a time-triggered as well as an event-triggered communication, such as MOST, FlexRay and Automotive Ethernet. In this context, this thesis assumes an architecture which provides predictability for safety-critical applications through a time-triggered execution scheme. Remaining resources might be used by less critical applications following an event-triggered execution scheme. In the work at hand, the focus lies on time-triggered scheduling. However, the proposed approaches might be extended to also take a concurrent event-triggered scheduling into account, i.e., through the use of suitable metrics during schedule synthesis. For instance, [TSPS12] proposes a schedule synthesis approach for TTEthernet, applying an objective function to minimize the end-to-end delay for event-triggered communication when the time-triggered schedule is generated. Furthermore, [MSKS13] investigates the timing behavior

for Automotive Ethernet, and gives recommendations on generating a time-triggered schedule, such that the latency for event-triggered communication is reduced.

The basic principles of time-triggered communication have been described in [KG94]. The paper proposes the Time-Triggered Protocol (TTP), a bus supporting time-triggered communication. Based on the TTP protocol, [KB03] introduces the Time-Triggered Architecture, a design methodology for a predictable E/E-architecture. The different challenges arising in the design of a time-triggered system are discussed, such as clock synchronization and specified communication interfaces, providing a temporal abstraction between components. Such a time-triggered architecture forms the basis for the approaches presented in this thesis.

1.7.1 Message Scheduling on the FlexRay Bus

Various work has been done on the message scheduling using a standard TDMA policy. For instance, [TC94] proposes a timing analysis to determine the worst-case response time analysis for event-triggered tasks communicating through a TDMA bus. The paper also considers the delays introduced by the protocol stacks in sending and receiving ECUs. In [HE05] Evolutionary Algorithms (EAs) are applied to optimize the slot assignment of tasks for resources applying a TDMA scheme. The paper proposes variation operators applicable for different EAs. Furhermore, several protocol specific approaches have also been proposed for different time-triggered protocols. For instance, [CV04] proposes a performance analysis for the Byteflight protocol [BPG00] which might be considered as the predecessor of FlexRay. In [PEP05] a schedule synthesis approach is proposed for heterogeneous systems consisting of a time-triggered TTP bus and an event-triggered CAN bus connected by a gateway. The paper is concerned with the packing of messages to frames such that all end-to-end delay requirements are meet. [ACZD94] proposes a schedule synthesis approach for a token ring network. The problem of assigning bandwidth to nodes is addressed with the goal of meeting all message deadlines. [SS07] proposes a schedule synthesis approach for message transmission with TTCAN. Several performance metrics are proposed, such as bandwidth utilization and message jitter. The paper also gives recommendations on desirable message properties. Finally, in [PEP04] four approaches for message to slot packing are proposed for a distributed system, communicating through TTP. For the implementation the paper proposes strategies based on greedy heuristics and SA.

Message scheduling on time-triggered buses follows a TDMA policy. Thus, each ECU can only transmit messages at predefined time intervals, referred to as slots. Each time-triggered protocol then defines additional constraints, for instance, the TTP protocol assigns exactly one slot to each ECU in one cycle while FlexRay supports multiple slots [Obe11]. FlexRay has very specific protocol properties such as constraints on the slot usage for different ECUs, making existing approaches for other time-triggered protocols or standard TDMA scheduling policies not directly applicable. A FlexRay specific approach becomes therefore necessary. Several work has been done addressing the FlexRay bus such as [GHN08, PPE$^+$08, LGTM09]. However,

most of these approaches are concerned with FlexRay 2.1. The latest FlexRay version 3.0 has introduced various changes, allowing a more efficient bandwidth usage. As existing approaches for version 2.1 are not applicable for FlexRay 3.0, new methodologies become necessary. The work at hand proposes a framework for FlexRay 3.0 that supports all new features, overcoming limitations of existing approaches for FlexRay 3.0 [SVG11, KPJ13]. We propose a generalized approach which is backwards compatible to FlexRay 2.1. A detailed discussion of work related to schedule synthesis for FlexRay is given in Section 3.1.1.

1.7.2 Modular Schedule Synthesis

The schedule integration framework presented in the work at hand addresses the problem of a holistic scheduling of tasks and messages in a time-triggered system, in the literature often also referred to as *static cyclic scheduling*. Various approaches have been proposed for holistic scheduling of time-triggered systems, see [LLP97, Obe11] for a detailed overview. A popular heuristic approach is list scheduling [CJG72]. List scheduling assigns the start-times to tasks based on an ordered list containing all tasks ready to be scheduled. Data-dependent tasks are only considered once all predecessor tasks have been scheduled. For instance, [BJM97] proposes a schedule synthesis for a distributed system communicating with an abstract communication bus. It applies a critical-path algorithm, i.e., tasks with the longest path in regards of execution-time are selected first. [THW02] proposes the heterogeneous earliest-finish-time algorithm, optimizing the schedule based on the task execution-times with the goal of minimizing the end-to-end delay. However, list scheduling approaches often include a mapping of tasks to processors or ECUs, while in the automotive domain the task mapping is often predefined, e.g., as sensors or actuators are attached to certain ECUs. In [EDPP00] a schedule synthesis approach for a distributed system communicating with TTP is proposed. The approach is based on list scheduling and assumes a predefined task mapping. As scheduling heuristics such as list scheduling might get stuck in local optima, creating suboptimal solutions, several approaches using metaheuristics have been presented to improve the obtainable solutions [Obe11]. For instance, [MSS89] presents an approach based on SA for single processor scheduling. One recent approach in [TSPS12] applies a tabu search-based algorithm to optimize the schedule for a system communicating with TTEthernet. The time-triggered schedule is optimized to minimize the end-to-end delay for event-triggered communication, sharing the communication bus with time-triggered messages. A drawback of heuristic and metaheuristic approaches is that for strongly constrained problems, e.g., if tight maximum end-to-end delays have been selected, the approach might struggle to determine a solution which satisfies all constraints and a backtracking is required [HLW$^+$14]. Schedule synthesis approaches using mathematical techniques allow to solve these problems, often with the goal of obtaining optimal results. For instance, [Ben96] proposes an ILP approach to schedule tasks on a heterogeneous multiprocessor system. In addition, [ZDGSV11] recently proposed a schedule synthesis approach based on an ILP formulation for task and message scheduling in a system communicating through a FlexRay bus.

The drawback of mathematical approaches is their limited scalability, making them unsuitable to schedule large numbers of tasks and messages. As a remedy, the work at hand proposes a modular approach which allows to integrate independently developed subsystem schedules in a global schedule. This *divide-and-conquer* approach allows to clearly improve the scalability of the schedule synthesis compared to generating a schedule from the scratch.

A modular design approach was developed in the area of *component-based design*. In this context, various work has addressed *hierarchical scheduling* which is concerned with the assignment of runtime budgets to subsystem schedulers [SL03, WT06]. Component-based design generally addresses scheduling of components on a single ECU, but might be extended to support distributed systems [KWL+12]. However, if component-based design is applied for distributed systems, distributed applications are split into several components. By contrast, schedule integration considers each application as an entity during schedule synthesis, facilitating changes to the schedule. Another body of related work can be found in the areas of *extensibility*, and *incremental scheduling*. The problem of extensibility is concerned with optimizing schedules to ease the integration of additional tasks and messages, while incremental scheduling addresses the problem of extending a schedule with minimal changes to the initial schedule. Examples for work addressing these problems are [PEPP04, WJP+05]. However, while incremental scheduling iteratively extends an existing schedule, schedule integration is concerned with a modular schedule synthesis, thus, with the integration of multiple individually generated schedules in a global schedule. Section 4.1 gives a more detailed discussion of work related to the schedule integration problem and how it differs from our approach.

1.7.3 Variant-aware Schedule Synthesis

Despite the great importance of variant management for the industry, variant management so far has only gained little attention by the scientific community. Beyond the design automation domain, several approaches for *product line* optimization have been proposed. A product line aims at developing a complete product family rather than several independent products. Thus, product line optimization aims at selecting a limited number of products such that the costs are kept minimal while achieving a high market share. [BFSS08] gives an overview of different approaches for product line optimization. A related problem is the determination of a *product platform* for a product line [NPP01], i.e., the selection of a common basis for different products, minimizing the cost. For E/E-architectures in the automotive domain, some work has been done concerning the management of different software product lines. For instance, [TH02] emphasizes on the importance of a systematic approach to variant management. An approach for modeling component variants using the intended features is proposed. [TDH11] investigates how a product line approach might be introduced for the software development with AUTOSAR. Finally, [MSR13] proposes the introduction of an *integrated feature modeling* in the management of different software product line variants, to document variant specific information such

as dependencies or restrictions. However, all these approaches are concerned with subsystems and do not address the whole E/E-architecture.

In [GGW+13] a system representation is proposed which allows to model all vehicle variants in a single system model referred to as *overspecified application model*. This application model contains the software components of all variants as well as the relations between each other. Based on this application model, [GGTL14] proposes a variant-aware Design Space Exploration (DSE). The proposed approach determines an individual architecture for each variant while determining an overall architecture selection, optimizing the reuse of components based on various objectives, such as overall cost. However, while the DSE proposed in [GGTL14] is concerned with the determination of variant architectures, the work at hand proposes a schedule synthesis approach to determine schedules for such variant architectures.

Work related to a variant-aware schedule synthesis can be found in the area of *multi-mode* scheduling. Multi-mode systems support several modes with different system configurations between which the system might switch at runtime, e.g., based on external events or fault detection. Schedule synthesis for multi-mode systems is therefore concerned with generating different system schedules, often with the goal of minimizing the switching delays [NNS+11, ACPF14]. By contrast, variant-aware schedule synthesis aims at minimizing the differences between variants, making multi-mode approaches not directly applicable. Besides this, there exist several approaches dealing with the concurrent scheduling of multiple graph-based applications in distributed systems in the context of heterogeneous distributed systems. Examples for such approaches are [IO98, ZS06]. The goal is to equally distribute the resources between the applications to minimize the overall makespan, i.e., schedule length. By contrast, we are concerned with generating multiple schedules sharing common tasks and messages that are deployed in different variants of the architecture. The multi-schedule synthesis proposed in this thesis is therefore among the first approaches addressing a variant-aware schedule synthesis. A detailed review of work related to a variant-aware schedule synthesis is given in Section 5.1.

1.8 Organization and Bibliographic Notes

Whereas this chapter introduces the background for the work described in this thesis, Chapter 2 gives a formal introduction to the schedule synthesis problems addressed here and introduces our system model. In Chapter 3 schedule synthesis approaches for the time-triggered FlexRay bus are presented. Section 3.1 presents different approaches for message scheduling on the FlexRay bus. The results presented in this chapter will appear in [SLC]. Section 3.2 presents an extension of this approach for holistic task and message scheduling, used as a baseline for the following chapters. Chapter 4 presents the schedule integration framework. First, the schedule integration methods are presented in Section 4.3, before metrics that allow to optimize the individual subsystem schedules are proposed in Section 4.4. The approaches presented in this chapter have been in parts published in [SLC13, SAW+14]. Chapter 5 presents a multi-schedule

approach considering different variants of an E/E-architecture. The results presented in this chapter have appeared in [SWS+16]. Finally, Chapter 6 concludes the thesis and discusses future research directions.

1.9 Publications

The approaches presented in Section 3.1 will appear as the following publication:

(1) Florian Sagstetter, Martin Lukasiewycz, Samarjit Chakraborty, *Generalized Asynchronous Time-Triggered Scheduling for FlexRay*, to appear in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

Parts of the contributions presented in Chapter 4 have appeared in the following publications:

(2) Florian Sagstetter, Sidharta Andalam, Peter Waszecki, Martin Lukasiewycz, Hauke Staehle, Samarjit Chakraborty, Alois Knoll, *Schedule Integration Framework for Time-Triggered Automotive Architectures*, in Design Automation Conference (DAC), San Francisco, USA, 2014.

(3) Florian Sagstetter, Martin Lukasiewycz, Samarjit Chakraborty, *Schedule Integration for Time-Triggered Systems*, in Asia and South Pacific Design Automation Conference (ASP-DAC), Yokohama, Japan, 2013.

The approach presented in Chapter 5 has appeared as the following publication:

(4) Florian Sagstetter, Peter Waszecki, Sebastian Steinhorst, Martin Lukasiewycz, Samarjit Chakraborty, *Multi-Schedule Synthesis for Variant Management in Automotive Time-Triggered Systems*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 35(4): 637-650, 2016.

The following publications are not directly part of this thesis but are generally related to the area of automotive E/E-architectures:

(5) Philipp Mundhenk, Florian Sagstetter, Sebastian Steinhorst, Martin Lukasiewycz, Samarjit Chakraborty, *Policy-based Message Scheduling Using FlexRay*, in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS), Delhi, India, 2014.

(6) Peter Waszecki, Florian Sagstetter, Martin Lukasiewycz, Samarjit Chakraborty, *Diagnosis-Aware System Design for Automotive E/E Architectures*, in International Symposium on Integrated Circuits (ISIC), Singapore, 2014.

(7) Martin Lukasiewycz, Sebastian Steinhorst, Sidharta Andalam, Florian Sagstetter, Peter Waszecki, Wanli Chang, Matthias Kauer, Philipp Mundhenk, Shanker Shreejith, Suhaib A Fahmy, Samarjit Chakraborty, *System Architecture and Software Design for Electric Vehicles*, in Design Automation Conference (DAC), Austin, USA, 2013.

(8) Florian Sagstetter, Martin Lukasiewycz, Sebastian Steinhorst, William R. Harris, Somesh Jha, Marko Wolf, Alexandre Bouard, Thomas Peyrin, Axel Poschmann, Samarjit Chakraborty, *Security Challenges in Automotive Hardware/Software Architecture Design*, in Design, Automation and Test in Europe (DATE), Grenoble, France, 2013.

(9) Martin Lukasiewycz, Sebastian Steinhorst, Florian Sagstetter, Wanli Chang, Peter Waszecki, Matthias Kauer, Samarjit Chakraborty, *Cyber-Physical Systems Design for Electric Vehicles*, in Euromicro Conference on Digital System Design (DSD), Cesme, Turkey, 2012.

2 System Representation

This chapter is intended to bridge the gap between the informal discussion on automotive E/E-architectures in the previous chapter and the formal system specification used for the approaches presented in the following chapters. First, we present a general system model for a time-triggered automotive architecture. Second, based on this model, schedule synthesis for a time-triggered system is introduced formally, defining the requirements for generating a schedule. Finally, we formally introduce the properties of the underlying hardware components and their constraints, e.g., for the FlexRay protocol.

The general notation and the symbols used to introduce the system description and throughout this thesis can be found in the List of Symbols (page xiii).

2.1 System Definition

The automotive architectures considered in this thesis are systems consisting of various networked resources. A system consists of multiple ECUs which are connected by at least one communication bus. Figure 2.1 illustrates an architecture consisting of four ECUs connected by a communication bus. Each ECU might also have individual links to sensors or actuators. If the system consists of multiple communication buses, single ECUs might be connected to multiple buses. These gateways allow to forward data from one bus to another bus. Hence, in accordance with current automotive architectures, an ECU can communicate with ECUs connected to other communication buses. The software executed on an ECU is represented by tasks and data is transmitted between ECUs in the form of messages. In the work at hand, tasks and messages are modeled as follows.

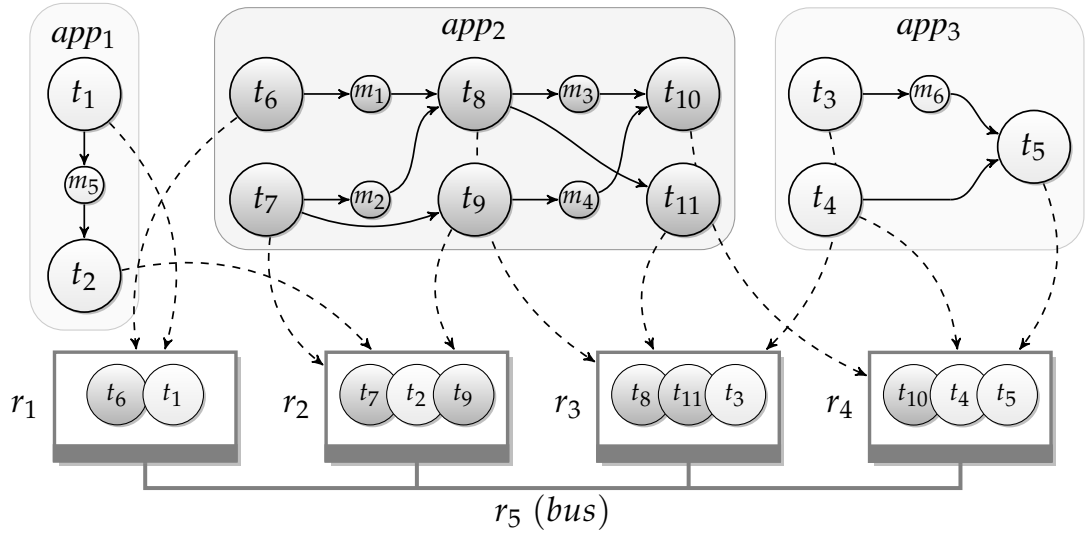


Figure 2.1: *System specification consisting of task graphs of three applications and their mappings to four ECUs connected by a communication bus.*

Definition 2.1.1 (process p)**:** *Both tasks executed on an ECU as well as messages transmitted via a communication bus are referred to as process p. Each process is defined by its execution-time τ_p and a period h_p. The execution-time equals the Worst Case Execution Time (WCET) for tasks executed on an ECU and the transmission time for a message sent on a communication bus, respectively. All processes are assumed to be executed strictly periodic.*

We assume that the WCET of all tasks and messages has been determined prior to applying a schedule synthesis, using tools such as aiT [FH04]. While not being in the focus of this work, sporadic processes might be transformed to periodic processes such that they can be considered in this model.

Automotive architectures implement various advanced functionality that is based on several distributed processes. For instance, a sensor process reads sensor data on one ECU which might be processed by a process on a second ECU. Hence, processes have data-dependencies which are considered by *applications*. Figure 2.1 illustrates a system executing three applications.

Definition 2.1.2 (application a)**:** *An application a is defined as a set of processes connected by data-dependencies. It is defined by a direct acyclic task-graph $G_a = (P_a, E_a)$, where P_a defines the set of processes and E_a the edges or data-dependencies, respectively, of the application. An application only contains processes which are weakly connected, i.e., have a data-dependency. While the periods of applications might differ, all processes of one application have the same period, due to the used activation model. An application commonly receives data from one or multiple sensors and activates one or multiple actuators. It is assumed that applications do not*

have cycles. However, an application having cycles can be adapted appropriately to break the cycles.

Each process is mapped to an ECU or communication bus on which it is executed. Figure 2.1 illustrates the mappings of processes to the respective ECUs.

Definition 2.1.3 (resource r)**:** *In the following, both ECUs and communication buses are referred to as resource r. It is assumed that each resource can only be utilized by one process at a time instant. In addition, each process can only be mapped to exactly one resource.*

While multi-core ECUs are not explicitly taken into account, this system model could be easily extended to support multi-core ECUs where each core is considered as an individual resource. This is also in accordance with the AUTOSAR standard where for each core an own entity of AUTOSAR OS is installed, allowing to statically configure each core [ZS14].

Based on the process, application and resource definitions, the whole system is represented by a *specification*, containing all required parameters for the scheduling problem. Figure 2.1 illustrates the specification of a small system including the process mappings.

Definition 2.1.4 (specification d)**:** *A specification defines the system description containing all applications A_d and contains the properties of each process. For simplicity, in the following, the processes of a specification might be referred to as P_d and all data-dependencies as E_d. Here, $P_d = \bigcup_{\forall a \in A_d} P_a$ and $E_d = \bigcup_{\forall a \in A_d} E_a$ holds. A specification also defines the mapping for each process p to a resource r.*

The presented system definition forms the basis for the schedule synthesis approaches presented in Section 3.2, Chapter 4 and Chapter 5. While the system definition also hold for the schedule synthesis approach in Section 3.1, there a simplified system description is used as the focus is on message scheduling.

2.2 Time-Triggered Scheduling

All approaches presented in this thesis are based on a time-triggered architecture, where time-triggered resources execute processes at time instants predefined by a schedule.

Definition 2.2.1 (time-triggered scheduling)**:** *A time-triggered schedule partitions the time into time-windows which are assigned to periodic processes. We define these time intervals by their start-time s_p and the execution-time τ_p for a process p. For a given start-time s_p, a process is executed during the time intervals $\mathbf{t}$ given by:*

$$s_p + n \cdot h_p \leq \mathbf{t} \leq s_p + \tau_p + n \cdot h_p, \quad \forall \mathbf{t} \in \mathbb{R}, n \in \mathbb{N}_0$$

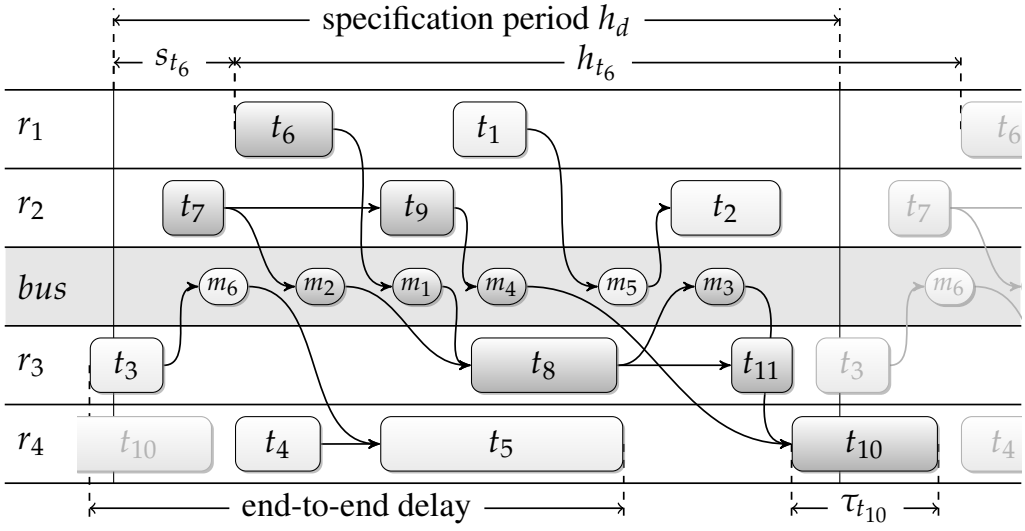


Figure 2.2: *Time-triggered schedule for the system specification in Figure 2.1.*

Hence, a time-triggered schedule reserves a time-window of the WCET τ_p with a temporal offset defined by the start-time s_p for each process p. This time-window is reserved for each periodic activation of the process based on the process period h_p.

Figure 2.2 illustrates a potential schedule for the specification in Figure 2.1 if all processes mapped to different resources are scheduled in a global schedule. A schedule synthesis approach must determine a feasible schedule such that only one process p utilizes the resource r at each point in time. To determine whether p utilizes r for a specific point in time t, the following function is defined:

$$\eta(p,\mathbf{t}) = \begin{cases} 1 & \forall \mathbf{t} : s_p + n \cdot h_p \leq \mathbf{t} \leq s_p + n \cdot h_p + \tau_p, n \in \mathbb{N}_0 \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Based on this Equation, the first requirement for a valid time-triggered schedule is defined as follows:

Requirement 2.2.1 (exclusive utilization)**:** *A resource can only be exclusively utilized by one process at a time instant. Non-preemptive time-triggered scheduling is assumed such that a resource r always completes the execution of one process before another process is started. With $r(p)$ denoting a function that returns the resource r to which the process p is mapped, the exclusive utilization requirement for one resource is defined as follows:*
$\mathbf{t} \in \mathbb{R}^+, p, \tilde{p} \in P_d, p \neq \tilde{p}, r(p) = r(\tilde{p}) :$

$$\eta(p,\mathbf{t}) + \eta(\tilde{p},\mathbf{t}) \leq 1 \tag{2.2}$$

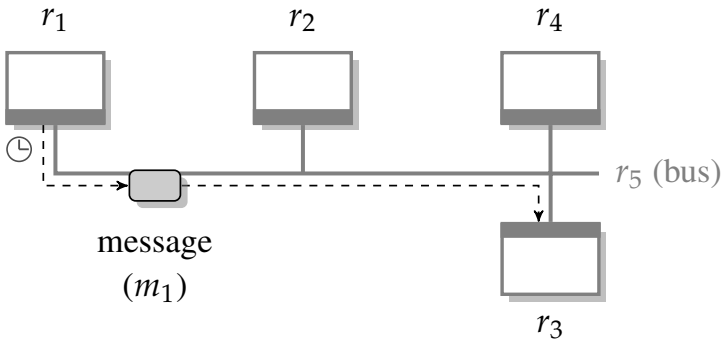


Figure 2.3: *For asynchronous scheduling each resource is scheduled individually. For this exemplary illustration, a message is transmitted from one ECU to another. To generate an asynchronous schedule, no knowledge of the sending or receiving tasks is required.*

Therefore, to create a feasible schedule, two processes $p, \tilde{p}$ mapped to the same resource r must not use r at the same point in time $\mathbf{t}$.

For time-triggered scheduling, two different approaches have to be distinguished: (1) *Asynchronous scheduling* and (2) *synchronous scheduling*. (1) For asynchronous scheduling no time synchronization is applied, i.e., each resource is scheduled individually. (2) By contrast, for synchronous scheduling all resources share a global time and all tasks and messages are executed in a time-triggered paradigm. A global schedule defines the start-times of all tasks and messages. In the following, first an introduction to asynchronous scheduling is given, before synchronous scheduling is introduced.

2.2.1 Asynchronous Time-Triggered Scheduling

For asynchronous scheduling, schedules for each resource are generated individually. Hence, a time-triggered schedule might be created for all processes on a resource without taking any data-dependencies into account. For instance, Figure 2.3 illustrates the transmission of a message on a time-triggered communication bus. The information about the sending ECU and the receiving ECU(s) might be required for the schedule synthesis, e.g., for message to frame packing, but no details about the sending or receiving tasks are necessary. Asynchronous scheduling is still predominantly used in the automotive industry, as subsystems are generally developed independently and asynchronous scheduling significantly reduces the integration efforts compared to synchronous scheduling. A common objective for asynchronous scheduling is the minimum bandwidth utilization of a bus. Section 3.1 presents an asynchronous time-triggered scheduling approach for the FlexRay bus.

2.2.2 Synchronous Time-Triggered Scheduling

For synchronous time-triggered scheduling, at runtime all tasks and messages are executed by a predefined schedule that is triggered by a global time. Automotive buses like FlexRay (see Section 2.3.2) or Automotive Ethernet (see Section 2.3.3) support time synchronization of ECUs, providing a global clock. For synchronous scheduling a global schedule considering all resources is generated. Figure 2.2 (page 30) illustrates a schedule for synchronous scheduling, defining the start-times for all tasks and messages. For data-dependent processes it is essential that data produced by a predecessor process is made available in a timely manner. Hence, during schedule synthesis additional constraints have to be taken into account to ensure that processes with data-dependencies are executed consecutively.

Requirement 2.2.2 (precedence)**:** *For data-dependent processes, it is essential that a data providing process terminates before a successor process $\hat{p}$ processing the data is started. With the edge e defining a data dependency and E_a defining all data-dependencies of an application a, it must hold that:*
$\forall e \in E_a, (p, \hat{p}) = e$*:*

$$s_p + \tau_p \leq s_{\hat{p}}$$

Hence, all processes must finish their execution before successor processes are started. This precedence constraint is of particular relevance when determining the end-to-end delay for an application.

Automotive architectures implement distributed applications running on a number of networked resources. Rather than defining deadlines for each single task, strict maximum end-to-end timing delays are defined for distributed applications [SRN+09]. Figure 2.2 (page 30) indicates the end-to-end delay for one of the applications. Note that for this illustrative example, only the end-to-end delay from sensor task t_3 to actuator task t_5 is indicated while the end-to-end delay from t_4 to t_5 was omitted for readability.

Definition 2.2.2 (source process p_{src})**:** *We refer to a process which does not have a data-dependency to a predecessor process, i.e., an incoming edge in the application task graph, as a* source process*. A source process is commonly a sensor task mapped to an ECU reading data from a sensor attached to it.*

Definition 2.2.3 (sink process p_{snk})**:** *We refer to a process which does not have a data-dependency to a successor process, i.e., an outgoing edge in the application task graph, as a* sink process*. A sink process is commonly an actuator task mapped to an ECU controlling an actuator connected to it.*

As our system model supports applications with multiple source and sink processes, an application might consist of multiple *paths*.

Definition 2.2.4 (path ϕ)**:** *A path ϕ defines a subgraph of an application from a source (sensor) to a sink (actuator) process. Each process in a path has only one predecessor and one successor, i.e., $\phi = \{p_1, p_2, .., p_n\}$. All processes in a path are connected by an edge in the application task graph. Sensor or source processes are obtained by the following function.*

$$\text{src}(\phi) = \{p | \exists p : \forall e \in \phi, \nexists e = (\check{p}, p)\}$$

Similarly, actuator or sink processes are obtained as follows.

$$\text{snk}(\phi) = \{p | \exists p : \forall e \in \phi, \nexists e = (p, \hat{p})\}$$

For each path we can now determine an end-to-end delay which must not exceed the *maximum end-to-end delay*.

Requirement 2.2.3 (maximum end-to-end delay θ_a)**:** *The maximum end-to-end delay of an application θ_a defines the longest permitted time frame from the start of the earliest source process to the finish of the latest sink process. With ϕ representing a path from a source to a sink process and $\Phi(E_a)$ returning all paths of the application a, the maximum end-to-end delay constrains an application as follows.*

$\phi \in \Phi(E_a), p_{src} = \text{src}(\phi), p_{snk} = \text{snk}(\phi)$*:*

$$\theta_a \geq (s_{p_{snk}} + \tau_{p_{snk}}) - s_{p_{src}}$$

Hence, the delay from the start of all source processes until all sink processes finish must not exceed the maximum end-to-end delay. The shortest feasible end-to-end delay is defined by the path with the largest total execution-time.

Based on these requirements, a schedule synthesis approach might now determine a schedule for a synchronous time-triggered system. For an efficient representation of the system schedule we use a periodic representation of the schedule.

Definition 2.2.5 (periodic schedule)**:** *A time-triggered schedule defines a start-time s_p for each process p (cf. Definition 2.2.1). As the system model considered here focuses on periodic processes, each process p is executed with its period h_p. Without loss of generality, the start-time s_p is therefore limited as follows.*

$$0 \leq s_p < h_p$$

Hence, a data-dependent process might have a lower start-time than its predecessor process. While this representation might require a short initialization phase when the system is started, it allows an efficient and comprehensive representation of a time-triggered schedule. A time-

triggered schedule for a specification d is then defined for the duration of a hyper-period h_d. The hyper-period is defined by the least common multiple of all processes in the specification:

$$h_d = \operatorname*{lcm}_{p \in P_d}(h_p)$$

Figure 2.2 (page 30) illustrates such a periodic schedule. For instance, task t_3 is started close to the end of the specification period h_d while the data dependent process m_6 is started at the beginning of the period.

A common objective for synchronous scheduling is a minimum end-to-end delay of distributed applications from sensor to actuator. The complexity of synchronous scheduling is significantly higher than for asynchronous scheduling. The approaches presented in Section 3.2, Chapter 4 and Chapter 5 address the problem of synchronous time-triggered scheduling.

2.3 Time-Triggered Communication Buses

The system definitions presented in the previous two sections considered the same properties for tasks and messages. However, for the communication on a time-triggered communication bus, additional message properties are required. The data exchanged between tasks is commonly referred to as *signals* in the automotive domain, while the data transmitted over a communication bus is referred to as message. Signals are therefore packed into messages before being transmitted on the bus. For our system model, we assume that signals are directly exchanged via memory if two data-dependent tasks are mapped to the same resource. If data is transmitted over a communication bus, we assume that all signals have already been packed into messages. While most approaches presented in this thesis are generic and the framework might be extended to support other automotive communication buses, the FlexRay bus and Automotive Ethernet are exemplary used. In the following, first a formal definition of TDMA is given which forms the basis for time-triggered communication protocols. Second, the FlexRay bus is introduced in more detail. Finally, the Ethernet bus properties defined in this thesis are presented.

2.3.1 Time-Division Multiple Access

Time-triggered communication protocols like FlexRay or Automotive Ethernet follow a TDMA scheme [Obe11] where the bandwidth is sliced into short time-intervals, i.e., time slots. A slot is exclusively assigned to a single ECU, preventing collisions, and has a fixed size. If an ECU does not transmit a message in an assigned slot, the slot remains unused. The slots assignment to ECUs are defined in a schedule prior to operation. As the schedule is known to all network participants, a message can be identified by the time slot it is sent in and no additional message header is required. Figure 2.4 shows a time-triggered communication following the TDMA scheme.

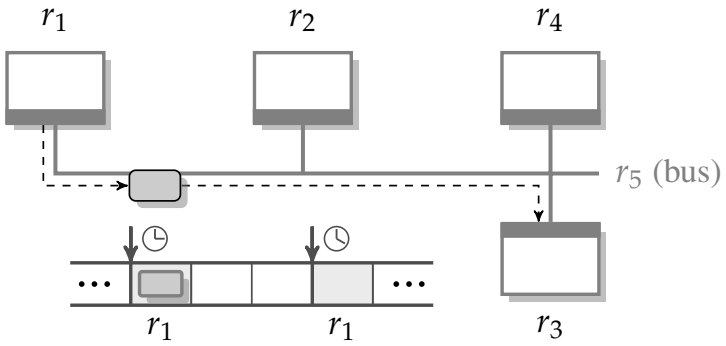


Figure 2.4: *Schematic illustration of a communication following a TDMA scheme.*

2.3.2 FlexRay

In the following, a formal introduction of the FlexRay protocol is given. A more detailed problem specific introduction is given in the respective sections (see Section 3.1 and Section 4.3.4). A FlexRay schedule is organized in c_{fr} communication cycles. A FlexRay communication cycle consists of four different segments as illustrated in Figure 2.5: (1) A *static segment* where messages are transferred in a predefined schedule, (2) an optional *dynamic segment* to transfer less critical and event-triggered data of variable length, (3) an optional *symbol window* for transferring special symbols, e.g., for bus initialization, and (4) the *network idle time* for the synchronization of network nodes. While all communication cycles have the same structure, the messages transmitted in each cycle might change. Once all cycles have been scheduled, the schedule is repeated.

The time-triggered static segment is commonly used for in-vehicle communication, while the event-triggered dynamic segment is used for diagnostics and configuration. Therefore, the focus of the work at hand lies on the static segment. The static segment follows a TDMA scheme and consists of a fixed number of n_{fr} slots of equal size. Each slot has a header, trailer and a payload segment, carrying data between 0 and 254 bytes. With a focus on the static segment, we define the following properties for a FlexRay bus.

Definition 2.3.1 (FlexRay bus)**:** *A FlexRay bus partitions the time of a periodically repeated schedule in n_{fr} time slots. At one time instant, a slot is exclusively used by the messages of a single sending ECU. All slots are of a equal size time interval of τ_{fr}. We assume that this time interval includes all additional delays introduced by the protocol. We also assume that the first time slot starts at time instant 0. A FlexRay cycle has the duration of h_{fr}, thus, a slot is available for message transmission with a period of h_{fr}, defining the minimal supported message period.*

One slot might contain multiple messages if the slot payload segment is not exceeded. As the payload size of a FlexRay slot l_{fr} is measured in bytes rather than its temporal duration, the process definition for messages is extended as follows.

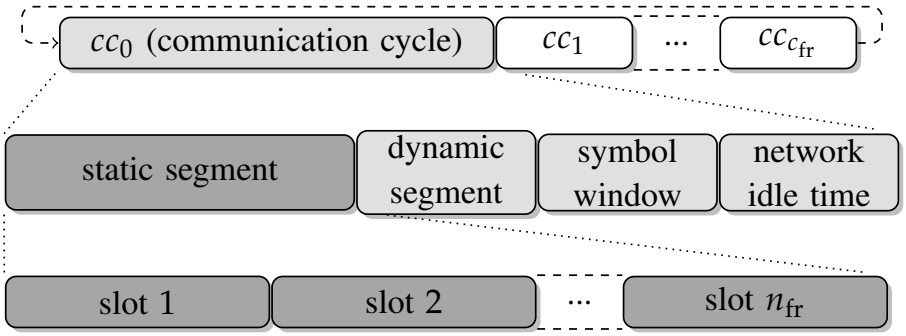


Figure 2.5: *Basic structure of FlexRay protocol organized in communication cycles with a detailed illustration of the static segment.*

Definition 2.3.2 (FlexRay - message)**:** *A message m is defined by its period and its size measured in bytes. The process definition in Definition 2.1.1 is therefore extended by a size parameter l_m.*

As a slot needs to be packed before transmission, the message start-times must equal the start-time of a slot. For synchronous time-triggered scheduling, we therefore define the following requirement for FlexRay messages.

Requirement 2.3.1 (FlexRay - slot assignment)**:** *Based on Definition 2.3.1, for a message m transmitted in slot $sl_m \in \{0, .., n_{\text{fr}} - 1\}$, the message start-time s_m needs to fulfill the following requirement.*

$$s_m = sl_m \cdot \tau_{\text{fr}}$$

Hence, a message can only be released at the start-time of the slot the message is transmitted in.

FlexRay versions. Several versions of the FlexRay protocol have been specified, introducing different constraints on the scheduling problem. Of relevance for the industry are version 2.1 and 3.0. FlexRay 2.1 is still the predominantly used FlexRay version for automotive systems, while FlexRay 3.0 is the most recent FlexRay version. All approaches presented in this thesis support both protocol versions. One major difference between both versions is that FlexRay 2.1 assigns a slot exclusively to one ECU. On the other hand, FlexRay 3.0 allows sharing of a slot between ECUs, such that a different ECU might transmit its messages in each cycle. Figure 2.6 illustrates this difference. Hence, for FlexRay 2.1 scheduling, we introduce the following requirement.

Requirement 2.3.2 (FlexRay 2.1 - exclusive slot usage)**:** *For FlexRay 2.1, a slot in the static segment is exclusively assigned to one ECU. Hence, for all cycles of the FlexRay schedule only one ECU is allowed to send its messages in one particular slot.*

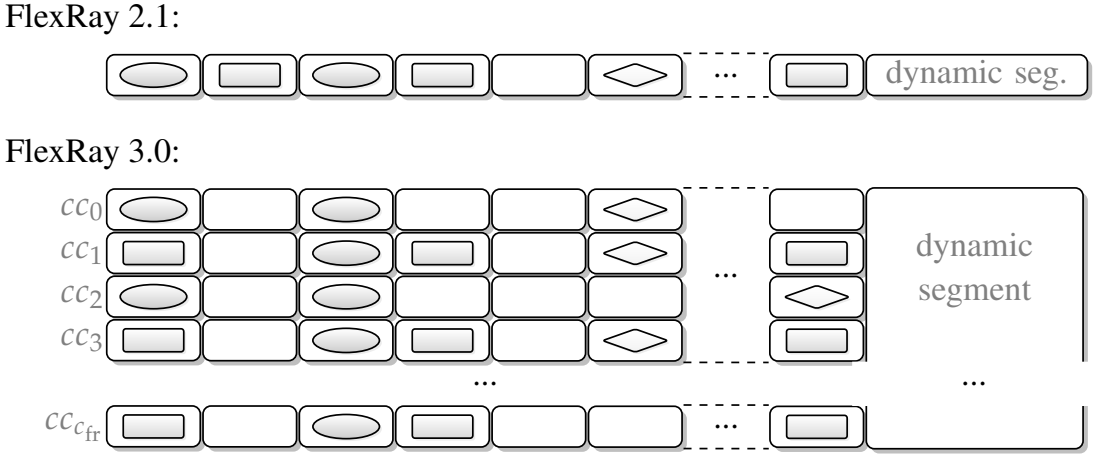
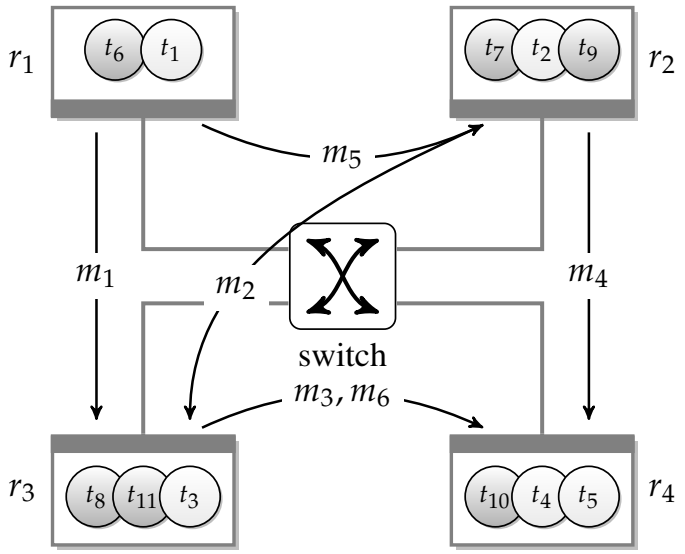


Figure 2.6: *Abstract illustration of slot to ECU assignment for FlexRay 2.1 and FlexRay 3.0, if three ECUs (⬭, ◇, ▭) send messages. For FlexRay 2.1, a slot is exclusively assigned to one slot, while for FlexRay 3.0, different ECUs can send their messages in the same slot for different cycles.*

2.3.3 Automotive Ethernet

Automotive Ethernet is currently still in the specification phase, as discussed in the introduction. However, various features and requirements have been specified, allowing to develop schedule synthesis methods that are compatible with a future Automotive Ethernet implementation. Automotive Ethernet will follow the set of standards defined as TSN [IEE15]. For time-triggered communication, a *Time-aware shaper* is introduced which prioritizes time-triggered communication over event-triggered communication based on a predefined schedule. TSN allows to define slots freely, supporting flexible message release times, rather than assigning messages to predefined slots as it is done for FlexRay. In our system model, we therefore consider Ethernet messages like processes scheduled on an ECU to which we assign a start-time. The message start-time and its transmission time then implicitly define the slot scheduled by the Time-aware shaper.

Automotive Ethernet is not implemented as a bus to which all ECUs are connected, but uses a switched network instead. Thus, an Ethernet network is realized by point-to-point connections between ECUs and switches. Ethernet also allows to connect multiple switches. For time-triggered scheduling this switched architecture signifies that multiple messages can be transmitted concurrently if the message paths in the network do not intersect [KCBQ14]. In addition, Ethernet supports a full-duplex communication, allowing an ECU to concurrently send and receive messages. Figure 2.7 illustrates the architecture from Figure 2.1 if realized by a central Ethernet switch. As switches today can handle multiple transmissions concurrently, for this example the central switch would allow to send the messages m_2, m_3 and m_5 concurrently.



link	messages
$r_1 \rightarrow sw$	m_1, m_5
$r_2 \rightarrow sw$	m_2, m_4
$r_3 \rightarrow sw$	m_3, m_6
$sw \rightarrow r_2$	m_5
$sw \rightarrow r_3$	m_1, m_2
$sw \rightarrow r_4$	m_3, m_4, m_6

Figure 2.7: *Illustration of an implementation of the architecture in Figure 2.1 based on full-duplex switched Ethernet. The table lists the messages sharing each link, connecting the switch with an ECU.*

The latest AUTOSAR standard for Automotive Ethernet offers support for Internet Protocol version 6 (IPv6) to define sending and receiving ECUs in the network. For the message transmission, the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) are supported. TCP is a stream-based protocol that requires to establish a connection between sender and receiver before data transmission. TCP includes various mechanisms to ensure a correct data transmission. However, these mechanisms also lead to a non-deterministic behavior, making TCP an unsuitable candidate for safety-critical applications [ZS14]. UDP is a message-based protocol which also supports multicast transmission as it is common for automotive applications [KCBQ14]. For the Ethernet communication, we therefore assume a UDP and IPv6-based communication which would be suitable for real-time applications. For these protocols, Ethernet introduces a significant overhead of 78 byte of header data for each transmitted message. Table 2.1 gives an overview of the header size. For 100 Mbit/s Ethernet the minimal message transmission time of a 64 byte UDP message on a link is then 11.36 µs. For switched Ethernet, each switch forwarding the message commonly adds an additional delay of 5 µs [ZS14]. In this thesis, an abstract model of the Ethernet bus is assumed which does not take the timing delays of each switch into account explicitly. Instead, to account for multiple switches and additionally introduced delays we assume a worst case transmission time for a message of 65 µs (for 64 byte message data) if not stated differently.

Table 2.1: *Structure of Ethernet message header.*

protocol	**overhead per message**
Ethernet 100 Mbit/s	30 byte[1]
IPv6 on Ethernet	$30 + 40$ =70 byte
UDP/IPv6 on Ethernet	$30 + 40 + 8$ =78 byte

For details see [ZS14]

[1] Sum of Preamble, MACs, VLAN tag, Type, FCS

3 Schedule Synthesis for FlexRay

In the previous chapter, a system model for a time-triggered automotive architecture was introduced. Based on this model, this section presents schedule synthesis approaches for a time-triggered system communicating via FlexRay. First, an approach for asynchronous time-triggered scheduling is proposed for the message scheduling on the FlexRay bus. Second, the message scheduling for FlexRay is extended with a concurrent task scheduling for synchronous scheduling. This extension also considers Automotive Ethernet.

To overcome the drawbacks of the predominant event-triggered CAN bus for automotive in-vehicle communication, a consortium of car manufacturers and suppliers developed the FlexRay protocol. It offers a hybrid topology layout with a high bandwidth, supporting time-triggered communication in the static segment, and event-triggered communication in the dynamic segment (see Section 2.3.2). With these characteristics, the FlexRay protocol is perfectly suitable for functions with very high safety demands such as drive-by-wire applications [YM09]. Car manufacturers like Audi or BMW already introduced the FlexRay bus in top-of-the-range series vehicles using version 2.1 of the protocol [BÖ07, KP08]. However, with the release of the FlexRay 3.0 standard in 2010 [Fle10], the FlexRay consortium introduced major changes, allowing a higher utilization of the bus. At the same time, the complexity of the protocol has increased and as a result new schedule synthesis techniques became necessary.

While schedule synthesis for FlexRay 2.1 has been extensively studied in literature, existing FlexRay 3.0 approaches have limitations regarding the obtainable results or do not consider all parameters required for a successful message scheduling[1]. As a remedy, this chapter proposes a systematic approach for FlexRay 3.0 schedule synthesis which at the same time is applicable

[1] A FlexRay schedule requires a slot, base-cycle and offset assignment for each message.

to version 2.1. In accordance with the current design approach in the automotive industry, the focus lies on message scheduling in the static segment, while the dynamic segment is commonly only used for configuration or diagnosis. As the FlexRay protocol has introduced additional constraints compared to traditional TDMA, like constraints on the slot usage for different senders, or the appended dynamic segment, TDMA scheduling policies are not applicable and a FlexRay specific approach is required.

Contributions and related work. This section presents a general, version-independent, scheduling framework for FlexRay. In Section 3.1 several approaches for message scheduling are presented. In this area, various work has been done for FlexRay 2.1 scheduling [GHN08, LGTM09, SS09b], however these approaches are not applicable to FlexRay 3.0. Existing FlexRay 3.0 approaches [SVG11, KPJ13] only consider frame to slot or message to frame packing, respectively, and omit a message offset assignment as in [DKK14]. By contrast, all approaches presented here are comprehensive. Our approaches not only obtain a message to slot assignment, but also assign a message offset within the slot, as required by the FlexRay specification. Offset assignment is required to efficiently use all new features of FlexRay 3.0, i.e., a variable cycle number. A single-stage ILP to determine an optimal solution and a heuristic with a good scalability are presented. Both approaches apply a direct message to slot packing to achieve minimal bandwidth utilization compared to a two-stage approach including frame packing. Furthermore, a multi-stage ILP approach is presented to improve the scalability of the ILP approach and to optimize legacy FlexRay 2.1 schedules. To evaluate these approaches, a Genetic Algorithm (GA) and a Simulated Annealing (SA) approach are also presented. The focus of these approaches is on message scheduling over the FlexRay bus, addressing the problem of asynchronous scheduling with the goal of a high bandwidth utilization. Section 3.2 then presents an extension of the single-stage ILP to support synchronous scheduling. The approach considers both the message scheduling on the FlexRay bus as well as the task scheduling on the ECUs. The proposed schedule synthesis allows to generate schedules for heterogeneous systems communicating with multiple FlexRay and Ethernet buses.

A more detailed introduction to the addressed problems including related work is given in Section 3.1 for asynchronous FlexRay scheduling, and in Section 3.2 for synchronous scheduling of heterogeneous networks.

3.1 Asynchronous Time-Triggered Scheduling

In the following, several schedule synthesis approaches for message scheduling with the FlexRay bus are presented. Thus, this section addresses asynchronous time-triggered scheduling suitable for robust control functions that are not affected negatively by additional delays like many automotive applications. Consequently, the proposed approaches are also in accordance with the current design approach in the automotive industry.

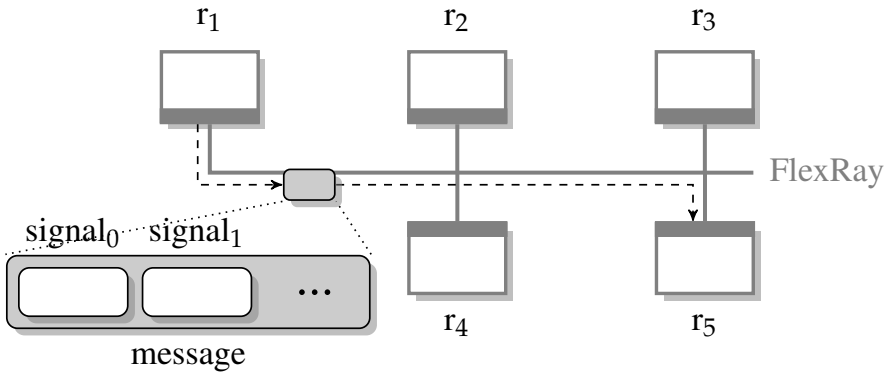


Figure 3.1: *FlexRay communication of two ECUs. Signals are packed into messages before being transfered over the bus.*

In the context of asynchronous scheduling, the literature often distinguishes between approaches for signals and messages. The data tasks of an application exchange are *signals* as shown in Figure 3.1. A *signal* could be physical data like the vehicle speed or a single control instruction. For message transmission on a communication bus, several *signals* of an ECU are then packed into a *message*. The system model applied in this thesis assumes that all signals are already packed into messages if tasks communicate using a communication bus like FlexRay[2]. Messages are measured in bytes as the smallest unit. A message might only be sent in a predefined time slot. Figure 3.2 gives an overview of the data structure of a FlexRay slot and the common terminology. Messages are packed into *frames* which equal the payload of a *slot*. A slot might contain multiple frames. FlexRay supports cycle multiplexing such that for each cycle, a slot may contain different messages. For FlexRay synthesis, generally two approaches exist. (1) A two-stage approach, first packing messages into frames before packing the frames into slots. (2) A direct packing of messages into slots, implicitly defining the frame packing. The two-stage approach reduces the problem complexity, while it might lead to suboptimal solutions with respect to the number of required slots. This section presents a single-stage ILP and a greedy heuristic providing a direct packing of messages to slots for a minimal bandwidth requirement. Furthermore, a multi-stage ILP applying the two-stage approach, which supports the integration of legacy systems, is presented.

Contributions. The scheduling problem addressed in this section is the determination of a schedule for a set of messages with the goal of a minimal bandwidth utilization. The schedule defines a slot, base-cycle, and offset for each message. Several techniques for scheduling the FlexRay static segment are presented that fulfill different requirements in terms of runtime and quality of results:

[2]Note that a direct packing of signals to frames compared to packing of messages to frames has no influence on the bandwidth requirements. A message solely groups signals, but does not introduce an overhead.

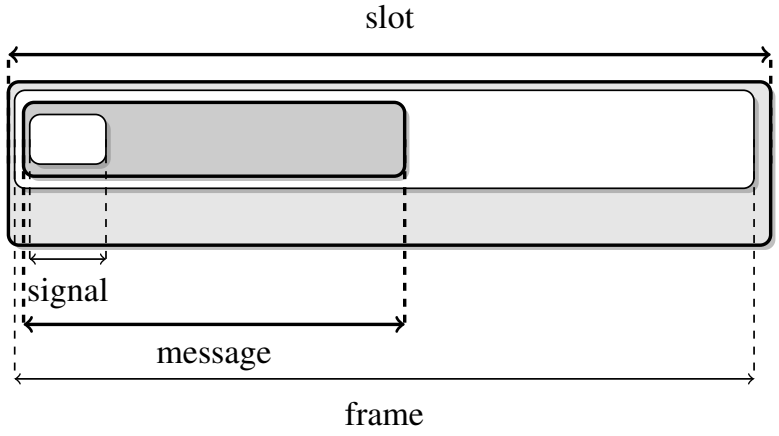


Figure 3.2: *Basic data structure of a FlexRay schedule. Signals are packed into messages before being sent by an ECU. Messages are sent in a frame assigned to a predefined time slot.*

- A single-stage ILP formulation to obtain an optimal schedule, suitable for problems with a moderate size and complexity.
- A multi-stage ILP approach which allows to first determine subsystem schedules before integrating them into a global schedule for the entire network. This is in particular relevant for legacy systems as testing and certification efforts are clearly reduced if an existing configuration is reused, instead of creating a new configuration from the scratch. The approach allows to convert or integrate existing FlexRay 2.1 schedules to FlexRay 3.0 schedules, optimizing their bandwidth usage.
- A greedy heuristic that determines schedules for full state-of-the-art in-vehicle networks, sending up to 1000 messages within few seconds while minimizing the required bandwidth.
- To evaluate the approaches, a GA and a SA approach were implemented. An extensive case study shows the benefits of our approaches compared to GA and SA-based approaches.

All approaches support both FlexRay 2.1 and 3.0, complying with the AUTOSAR standard used in the automotive industry for all FlexRay implementations [AUT14]. A FIeld Bus Exchange Format (FIBEX) import and export allows to integrate the approaches in existing tool-chains. FIBEX is the standard data/information exchange format to define and configure automotive networks [ASA10].

Table 3.1 gives an overview of the capabilities of all approaches. The ILP produces optimal results, but for FlexRay 3.0 it does not scale well and scheduling 130 messages might already take more than 24 hours as the results indicate. The multi-stage ILP approach overcomes this limitation through integrating independently created subsystem schedules into a global schedule. However, its results are not globally optimal. Finally, the greedy heuristic determines close

Table 3.1: *Characteristics of approaches proposed for FlexRay scheduling.*

method	bandwidth usage	scalability	extensibility/ modularity
Single-stage ILP	++[1]	- -	- -
Multi-stage ILP	○[2]	+	++
Greedy Heuristic	++	++	- -
Genetic Algorithm	+	+	- -
Simulated Annealing	+	+	- -

[1] optimal approach, determining minimal bandwidth usage

[2] average bandwidth utilization

to optimal results and scales well as our case studies show. While both the single-stage ILP and the greedy heuristic require generating a schedule from the scratch, the multi-stage ILP approach supports a modular design approach. It allows to integrate and extend schedules created by the two other approaches. For completeness, a GA and a SA approach are also presented. However, for large message sets, both are outperformed by the greedy heuristic which requires up to 7 % less bandwidth while being about 3 orders of magnitude faster. A large case-study consisting of 420 test-cases and a realistic test case of a full in-vehicle network are presented to give evidence of the efficiency of the proposed scheduling approaches. At the same time, the results show the possible advantages of FlexRay 3.0 over FlexRay 2.1 in terms of the utilization of the bus.

Outline. Section 3.1.1 first gives a detailed discussion of related work. Section 3.1.2 gives a detailed problem description and introduces our framework. Section 3.1.3 presents the single-stage ILP formulation, while Section 3.1.4 proposes the multi-stage ILP approach. Section 3.1.5 introduces the greedy heuristic approach including a sorting strategy based on a GA and a SA approach. Finally, Section 3.1.6 evaluates our approaches using an extensive performance analysis and a realistic case study.

3.1.1 Related Work

The FlexRay protocol specification was developed by the *FlexRay consortium* including *Bosch, BMW, General Motors* and *Volkswagen*. The protocol was finalized in 2010 with the release of FlexRay 3.0. After its first implementation in a series-production vehicle with the BMW X5 in 2006 [BÖ7], the FlexRay bus is currently becoming an integral part of the in-vehicle networks of top-of-the range cars [BPS08, KP08]. All current series-production vehicles using the FlexRay bus are compliant with the FlexRay AUTOSAR Interface Specification [AUT14].

Recent publications cover various topics regarding FlexRay networks. In [BKPS07], an introduction to the configuration of FlexRay applications is given. An optimization method to define optimal parameters for the static segment and the communication cycle can be found in [PS11]. In [Cen06] and [HBC+07], a timing and performance analysis of the FlexRay protocol is proposed with a focus on the dynamic segment. Several schedule optimization methods have been proposed for the dynamic FlexRay segment like in [PPEP07, SS09a, SYP+09]. However, here the focus is on the static segment as the dynamic segment is commonly only used for diagnosis and configuration.

Approaches for synchronous scheduling in the static FlexRay segment have been presented in [ZDGSV11] and [LSGC12], applying a concurrent task and message scheduling. However, the approaches presented in this section are concerned with asynchronous scheduling. Section 3.2 then presents an extension of the optimal ILP approach for synchronous scheduling.

In the automotive domain, ECUs, their functions, and their schedules are currently still largely developed and determined independently, preventing a synchronous scheduling. As a result, many applications and robust control functions do not require synchronization of tasks such that asynchronous scheduling may be applied. This reduces the complexity of the scheduling and integration efforts significantly. [DMTT05, DTT08] present an approach for asynchronous FlexRay scheduling using a GA. The approach also takes the timing delay from sending to receiving tasks into account. However, as the authors state in [Din10], GAs do not scale for non-trivial problems, and therefore propose a hybrid approach using a mix of a bin-packing algorithm and a GA. In [SS09b] a two-stage ILP approach is proposed. It packs signals to frames in a first step before determining a schedule from the frames. Two scheduling approaches, complying with the AUTOSAR standard, can be found in [GHN08] and [LGTM09]. In [GHN08], FlexRay schedules are obtained by a heuristic, assuming one signal per frame and, thus, leading to inferior bandwidth utilization. In [LGTM09], the authors present a schedule synthesis that considers the packing of messages to slots based on the bin-packing problem [LMV02]. In summary, all discussed approaches are tailored to FlexRay 2.1 and are not applicable for FlexRay 3.0. As a remedy, a generalized approach that obtains asynchronous schedules and is applicable to the different versions of FlexRay is presented.

Approaches supporting FlexRay 3.0 features have been proposed in [HLW+14] and [DK14], addressing the problem of holistic scheduling with FlexRay. [HLW+14] proposes heuristic approaches based on list scheduling while [DK14] applies a transformation to the strip packing problem to schedule the messages. While both approaches support slot sharing as introduced by FlexRay 3.0, they do not consider a variable cycle number. A variable cycle number might allow to clearly reduce the bandwidth requirements as our results show. An approach for asynchronous FlexRay 3.0 scheduling is presented in [SVG11]. In contrast to the work at hand, the paper only considers a packing of frames to slots, and assumes the packing of messages to frames was done in a previous design step. It is therefore not applicable to many relevant automotive scenarios that rely on the AUTOSAR specification where a given frame packing is not assumed. [KPJ13] presents an approach for packing signals to frames, supporting multi-

ple periods in a frame as supported by FlexRay 3.0. To generate the final schedule, it relies on the scheduling analysis proposed in [PPE+08]. While these approaches reduce the problem complexity, they also lead to clearly worse results than an implicit frame packing that is defined by scheduling of messages directly into slots. Finally, in [DKK14] an approach for holistic scheduling for a system connected by a FlexRay bus is presented. The paper proposes an ILP approach which considers a direct packing of messages to slots suitable for FlexRay 3.0 scheduling. However, [DKK14] does not consider a message offset assignment. Offset assignment becomes necessary for FlexRay 3.0 where a variable cycle number allows to send messages with repetitions that only share 1 as greatest common divisor in one slot, as discussed in detail in Section 3.1.2.2. As a remedy, this section proposes several approaches that comply with the AUTOSAR specification and considers a direct message to slot packing for the single-stage ILP and the greedy approach, obtaining the best possible utilization of the bus. In addition, the multi-stage ILP allows to convert legacy FlexRay 2.1 schedules to FlexRay 3.0 schedules with reduced bandwidth requirements. All approaches include a message offset assignment which has been omitted by previous work.

3.1.2 FlexRay Scheduling

This Section first introduces the general scheduling in the FlexRay static segment and its notation. Second, a depiction of the differences between FlexRay 2.1 and 3.0 is given. Finally, the framework is introduced.

3.1.2.1 FlexRay Parameters

Extending the high-level introduction in Section 2.3.2, in the following, a general introduction to FlexRay scheduling is given which applies to both FlexRay 2.1 and 3.0. All major car manufacturers follow the AUTOSAR standard for their FlexRay implementations in series-production vehicles. To increase the bandwidth usage of the FlexRay bus, AUTOSAR suggests cycle multiplexing. Cycle multiplexing allows to alternate the messages sent in one slot from cycle to cycle to increase the utilization, for instance, see messages m_{01} and m_{02} alternating in Figure 3.3.

FlexRay requires the configuration of various parameters. The number of communication cycles c_{fr}, the cycle duration h_{fr}, the payload size of a static slot l_{fr} in bytes, and the number of available static slots n_{fr} in one cycle. A message m is defined by its size l_m and its period h_m. To schedule a set of messages $m \in M$, FlexRay requires that all message periods are a multiple of the communication cycle duration h_{fr}. If this is the case, the message period can be represented by the message repetition $\rho_m = \frac{h_m}{h_{\text{fr}}}$. Hence, the message repetition ρ_m represents the message period as a multiple of the FlexRay cycle duration. Given a message m with its size l_m and repetition ρ_m, scheduling is formally defined as follows. To schedule m, FlexRay requires ρ_m to be a divisor of the number of cycles c_{fr} to allow a periodic occurrence. If this is not the case, oversampling becomes necessary, i.e., a smaller value of ρ_m has to be chosen. To

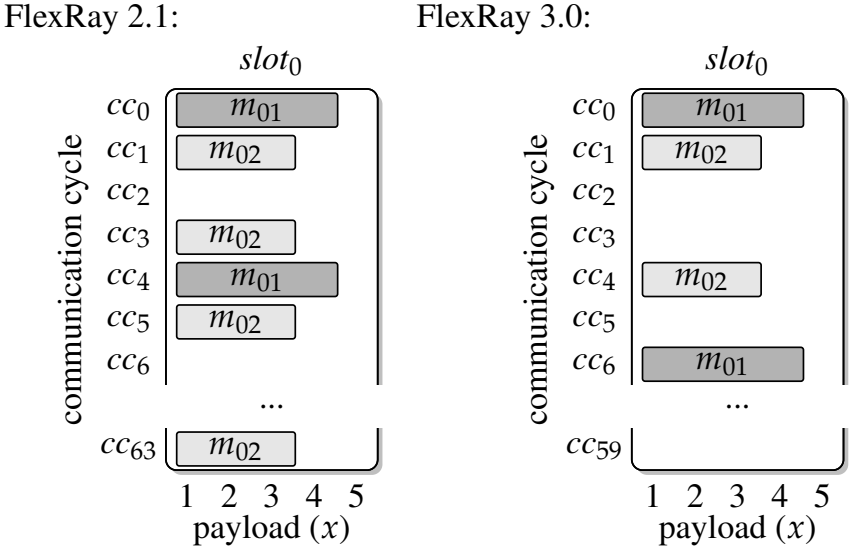


Figure 3.3: *Comparison of version 2.1 and 3.0 of FlexRay. FlexRay 2.1 limits the cycle number c_{fr} to 64 while 3.0 allows to customize c_{fr}. Selecting a suitable cycle number with FlexRay 3.0 might prevent oversampling and save bandwidth. Here, the messages m_{01} and m_{02} can be sent with their initial periods $\rho_{m_{01}} = 6$ and $\rho_{m_{02}} = 3$ for $c_{\text{fr}} = 60$ cycles while oversampling is required for $c_{\text{fr}} = 64$ as it is not divisible by 3 or 6.*

schedule a message m, three parameters need to be determined. (1) The slot sl_m in which m is scheduled. (2) The base-cycle b_m that indicates the first cycle in which m is scheduled. (3) The x-offset x_m, indicating at which position in the slot the message m is scheduled. For instance, in Figure 3.3 m_{02} is scheduled in slot 0 with a base-cycle 1 and x-offset 0. Thus, a message m is scheduled in all cycles cc_i with:

$$i = (b_m + \rho_m \cdot n) \bmod c_{\text{fr}}, n \in \mathbb{N}_0 \tag{3.1}$$

This scheduling has to be performed such that no messages intersect. Two messages m and $\tilde{m}$ scheduled in the same slot, at the same payload position, do not intersect if they never share the same cycle:

$\forall i, j \in \mathbb{N}_0$:

$$(b_m + \rho_m \cdot i) \bmod c_{\text{fr}} \neq (b_{\tilde{m}} + \rho_{\tilde{m}} \cdot j) \bmod c_{\text{fr}} \tag{3.2}$$

Schedules for single FlexRay slots are illustrated in Figure 3.3 and 3.4. The representation is derived from Figure 2.5 (page 36) through introducing a row for each cycle on the y-axis. Hence, b_m is the offset on the y-axis and x_m is the offset on the x-axis within a slot.

3.1.2.2 FlexRay Versions

The FlexRay 3.0 standard introduces several changes (in comparison to version 2.1) to the FlexRay protocol. In particular, for scheduling the static segment, the following two major changes are highly relevant and might improve the utilization of the bus. (1) The number of cycles c_{fr} is variable and (2) slots can be shared by different ECUs.

(1) The number of cycles c_{fr} of a FlexRay schedule is configured at design time. For FlexRay 3.0, c_{fr} can be any even number of cycles between 8 and 64. A message can only be scheduled with a repetition ρ_m that is a common divisor of c_{fr}, hence $c_{\text{fr}} \bmod \rho_m = 0$ must hold. As example for $c_{\text{fr}} = 60$, a message m might be sent with repetitions of $\rho_m \in \{1,2,3,4,5,6,10,12,15,20,30,60\}$ cycles. By contrast, for version 2.1 of FlexRay, the number of cycles of a FlexRay schedule is fixed to $c_{\text{fr}} = 64$, defining $\rho_m \in \{1,2,4,8,16,32,64\}$. The variable cycle number of FlexRay 3.0 might allow a higher flexibility in terms of available repetition values. Hence, a lower bandwidth utilization can be achieved by preventing oversampling. Figure 3.3 illustrates how two messages need to be oversampled for $c_{\text{fr}} = 64$ in comparison to $c_{\text{fr}} = 60$. For this example, scheduling with $c_{\text{fr}} = 60$ leads to over 30 % less bandwidth utilization compared to the $c_{\text{fr}} = 64$ of FlexRay 2.1. The unused slot space might then be used to schedule other messages and reduce the total number of slots. However, while $c_{\text{fr}} = 64$ only supports message repetitions which are a multiple of 2, a variable cycle number allows to send messages with repetitions that only share 1 as greatest common divisor in one slot. Hence, while message offsets might be trivially defined for FlexRay 2.1, version 3.0 requires a specific offset assignment[3].

(2) Additionally, while for FlexRay 2.1 a slot is exclusively assigned to one ECU to send messages, version 3.0 allows slot sharing between different ECUs. Thus, for each communication cycle, a different ECU might send its messages in a specific slot. Figure 3.4 shows how three messages sent by two ECUs can be scheduled in a single slot with FlexRay 3.0 while version 2.1 requires two slots to schedule the same messages. While this new feature allows to use slots more efficiently, the problem complexity is clearly increased as a concurrent scheduling of all ECUs becomes necessary. Hence, while the schedule synthesis might be applied for each ECU independently for version 2.1, FlexRay 3.0 requires a global scheduling approach that requires a more complex optimization model.

3.1.2.3 Schedule Synthesis Framework

We propose a framework to determine a FlexRay schedule for a set of messages. The framework allows a schedule synthesis for FlexRay 3.0, taking into account all its new features. At the same time, all approaches are backwards compatible to FlexRay 2.1. The main optimization objective is the minimization of required slots which is a common approach for asynchronous scheduling since it implicitly improves the utilization of the FlexRay bus.

[3]For two messages m and $\tilde{m}$ with $1 < \rho_m < \rho_{\tilde{m}}$ and $\gcd(\rho_m, \rho_{\tilde{m}}) = 1$, Eq. (3.2) is not satisfied for any base-cycle combination. Therefore, the offset assignment must ensure that m and $\tilde{m}$ are placed after each other.

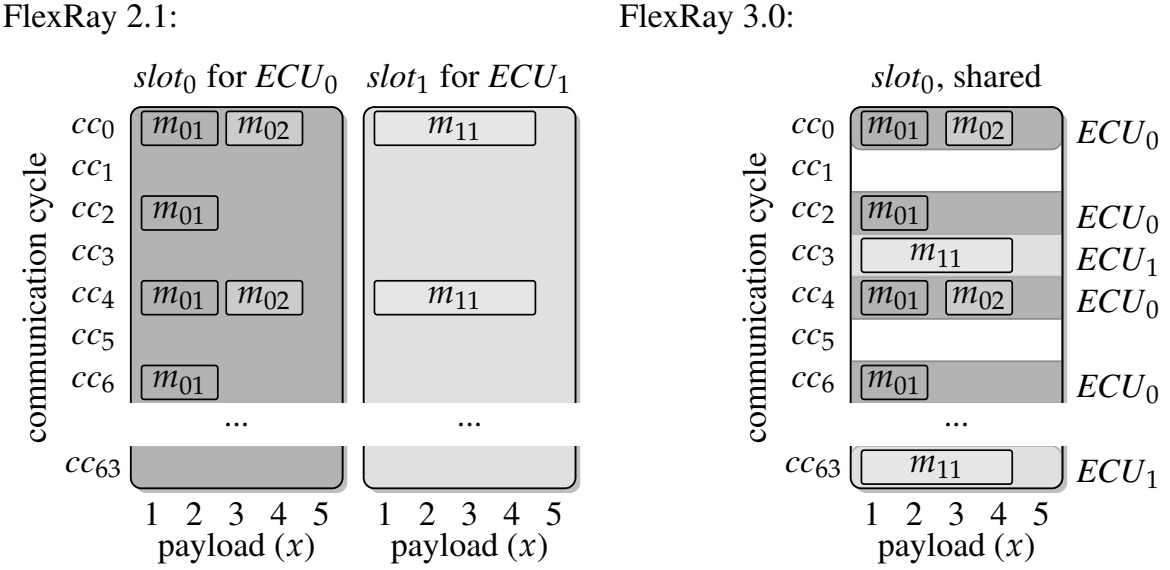


Figure 3.4: *Comparison of version 2.1 and 3.0 of FlexRay. FlexRay 2.1 requires a slot to be explicitly assigned to one ECU while 3.0 supports multiple ECUs per slot. Slot sharing allows to further reduce the bandwidth usage with FlexRay 3.0.*

The framework is illustrated in Figure 3.5. As input, the optimization algorithm requires the parameters of the FlexRay bus such as the communication cycle duration h_{fr}, the available payload in a static slot l_{fr}, and the number of static slots n_{fr}. The set of messages M defines a size l_m and repetition ρ_m for each message m. To comply with design tools used in the automotive industry, the framework supports the FIBEX data/information exchange format [ASA10]. The XML-based FIBEX format is the common exchange format to describe automotive networks, supporting various protocols like CAN or FlexRay. It specifies the network topology, ECU information, the FlexRay network configuration, message[4] and frame definitions. Based on this information, our framework determines a schedule and exports the results to a FIBEX file. This allows to easily integrate our schedule synthesis with other tools in the automotive domain.

3.1.3 ILP-based Optimal Schedule Synthesis

This section presents a single-stage ILP formulation to determine a message to slot assignment. It determines a schedule, using a minimal number of slots for a given set of messages, implicitly minimizing the bandwidth usage. As the FlexRay specification also requires a message offset within the slot, this Section also presents an ILP approach to determine suitable offsets in a second step. Figure 3.6 illustrates this approach.

[4]Messages are defined in the form of AUTOSAR Protocol Data Units.

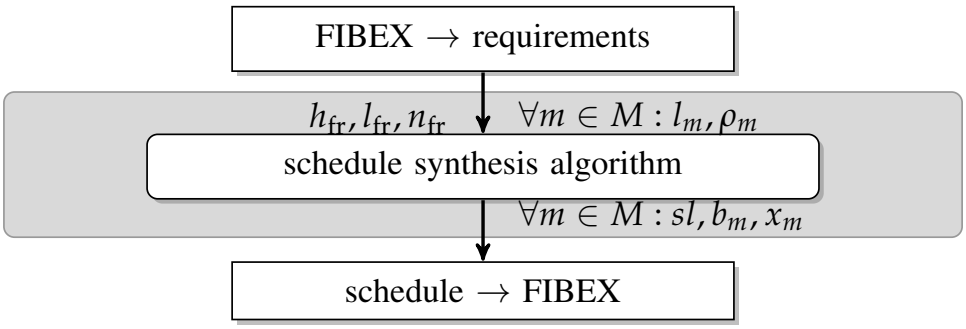


Figure 3.5: *Framework for FlexRay scheduling. For a set of requirements, a feasible schedule is determined. These user defined requirements include the FlexRay parameters and the message set sent over the FlexRay bus.*

3.1.3.1 Single-stage ILP for Slot and Base-Cycle Assignment

The ILP determines a slot sl and a base-cycle b_m for each message m in a given set M. For FlexRay 3.0, this schedule is determined for all ECUs in a global ILP. If slot sharing is not possible like in FlexRay 2.1, the ILP is applied to each ECU separately and the allocated slots are combined to the global schedule afterwards. The ILP is based on the following constants, as introduced in detail in Section 3.1.2:

- $sl \in Sl$ - denotes index for slot.
- $m \in M$ - message, where M denotes the set of all messages.
- $\check{r} \in \check{R}$ - ECU, where $\check{R}$ contains all ECUs sending at least one message.
- $M(\check{r}) : R \rightarrow \mathcal{M}$ - function returning all messages sent from $\check{r}$.
- $c \in \{0, .., c_{fr} - 1\}$ - cycle, where c_{fr} denotes the number of cycles of the FlexRay schedule.
- l_{fr} - denotes length of a slot in byte.
- ρ_m - repetition rate representing the period of message m as multiple of FlexRay cycles.
- l_m - size of message m in byte.

The ILP uses the following variables:

- $\bar{\mathbf{y}}_{sl} \in \{0, 1\}$ - binary variable indicating if a slot sl is allocated
- $\bar{\mathbf{z}}_{(m,sl,b)} \in \{0, 1\}$ - binary variable indicating if message m is scheduled in slot sl with base-cycle b

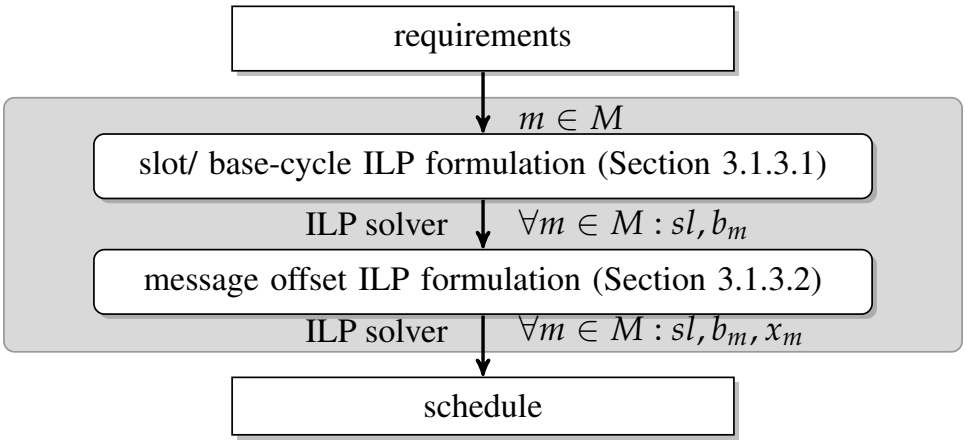


Figure 3.6: *Optimization flow determining a schedule with minimal bandwidth utilization. A single-stage ILP approach first assigns a slot and base-cycle to a message, before the message offset is determined in a second step.*

- $\bar{\mathbf{v}}_{(\check{r},sl,c)} \in \{0,1\}$ - binary variable indicating ownership of an ECU $\check{r}$ for slot sl in cycle c

The ILP formulation minimizes the following objective:

$$\min \sum_{sl \in Sl} \bar{\mathbf{y}}_{sl} \tag{3.3}$$

Hence, the number of allocated slots is minimized. Here, an upper bound of required slots is defined by the cardinality of Sl. Based on this objective, the following constraints determine a schedule defining a slot and base-cycle for each message m:
$\forall m \in M$:

$$\sum_{sl \in Sl} \sum_{b=0}^{\rho_m - 1} \bar{\mathbf{z}}_{(m,sl,b)} = 1 \tag{3.4}$$

$\forall sl \in Sl, c = \{0, ..., c_{\text{fr}} - 1\}$:

$$\sum_{\substack{m \in M \\ b = c \bmod \rho_m}} \bar{\mathbf{z}}_{(m,sl,b)} \cdot l_m \leq l_{\text{fr}} \tag{3.5}$$

$\forall sl \in Sl, m \in M, b = \{0, ..., \rho_m - 1\}$:

$$\bar{\mathbf{y}}_{sl} - \bar{\mathbf{z}}_{(m,sl,b)} \geq 0 \tag{3.6}$$

$\forall sl \in Sl, c = \{0, ..., c_{\text{fr}} - 1\}$:

$$\sum_{\check{r} \in \check{R}} \bar{\mathbf{v}}_{(\check{r},sl,c)} \leq 1 \tag{3.7}$$

$\forall sl \in Sl, c = \{0,..,c_{\text{fr}}-1\}, \forall \check{r} \in \check{R}, m \in M(\check{r}), b = c \bmod \rho_m$:

$$\bar{\mathbf{v}}_{(\check{r},sl,c)} - \bar{\mathbf{z}}_{(m,sl,b)} \geq 0 \tag{3.8}$$

Constraint (3.4) ensures that each message is scheduled in exactly one slot with a specific base-cycle. To ensure that messages scheduled in a slot do not exceed the slot size, the sum of all message sizes in one cycle must not be larger than the slot length as stated in Constraint (3.5). Furthermore, a message can only be scheduled in slots that have been allocated as defined in Constraint (3.6). Constraint (3.7) ensures that each cycle in a slot can only be filled by messages from not more than one ECU. If a message is scheduled in a specific cycle, its sender needs to be assigned as cycle owner as stated in Constraint (3.8).

For FlexRay 2.1 scheduling, $c_{\text{fr}} = 64$ is fixed and a slot is exclusively assigned to one sender. Hence, the ILP is executed for each ECU separately, and Constraints (3.7) and (3.8) are omitted.

3.1.3.2 ILP for Message x-Offsets

The ILP presented in the previous section assigns a slot and a base-cycle to all messages. As several messages can be scheduled in one slot in the same cycle, additionally an offset x_m needs to be assigned to each message m. To determine appropriate values for the message offset, a second ILP for each slot is formulated which uses the previously calculated allocation of m to sl and b_m to determine:

- $\mathbf{x}_m \in \mathbb{N}_0$ - integer-variable for x-offset of message m
- $\bar{\mathbf{x}}_{(m,\tilde{m})} \in \{0,1\}$ - binary variable indicating that message m is placed before message $\tilde{m}$ in the slot

The ILP is applied to each slot separately and formulated as follows:
$\forall m \in M_{sl}$:

$$0 \leq \mathbf{x}_m \leq l_{\text{fr}} - l_m \tag{3.9}$$

$\forall m, \tilde{m} \in M_{sl}, \exists c \in \{0,..,c_{\text{fr}}-1\}, c \bmod \rho_m = b_m \wedge c \bmod r_{\tilde{m}} = b_{\tilde{m}}$:

$$\mathbf{x}_m + l_m \leq \mathbf{x}_{\tilde{m}} + l_{\text{fr}} \cdot \bar{\mathbf{x}}_{(m,\tilde{m})} \tag{3.10}$$

$$\mathbf{x}_{\tilde{m}} + l_{\tilde{m}} \leq \mathbf{x}_m + l_{\text{fr}} \cdot \bar{\mathbf{x}}_{(\tilde{m},m)} \tag{3.11}$$

$$\bar{\mathbf{x}}_{(\tilde{m},m)} + \bar{\mathbf{x}}_{(m,\tilde{m})} = 1 \tag{3.12}$$

The ILP first defines appropriate bounds for the x-offset for each message in Constraint (3.9). If two messages are sent in at least one cycle together, it holds that either the first message is transmitted before the second one starts or vice versa, see Constraints (3.10)

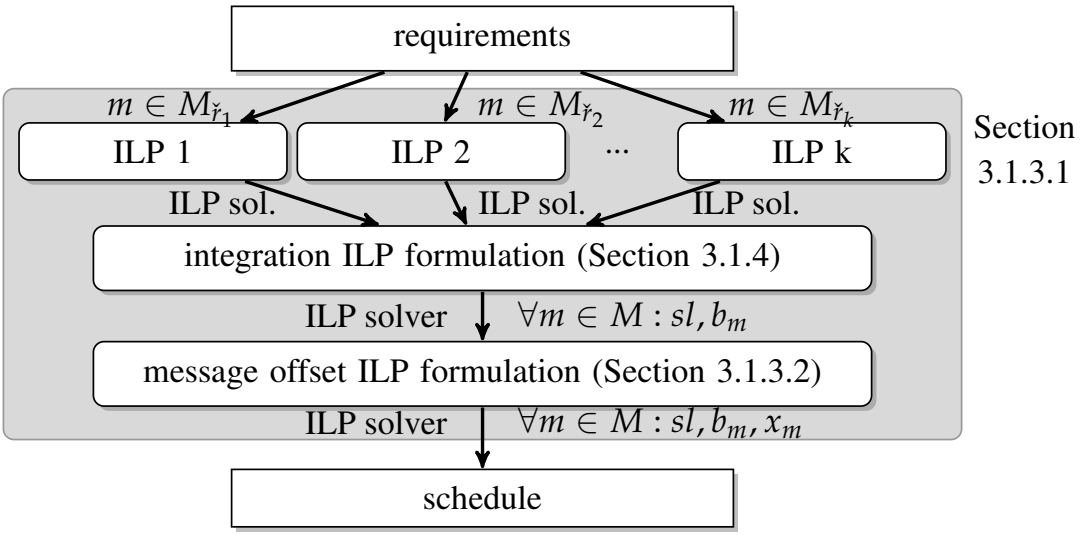


Figure 3.7: *Optimization flow for the multi-stage ILP approach. First, the single-stage ILP approach is applied for each subsystem individually, before the integration ILP combines the results to a global schedule. Finally, the message offsets are determined.*

and (3.11). Here, the binary variables $\bar{\mathbf{x}}_{(m,\tilde{m})}$ and $\bar{\mathbf{x}}_{(\tilde{m},m)}$, respectively, are used as switch variables. Only one of the switch variables can be active as stated in Constraint (3.12). As the ILP is applied to each slot individually, the problem size is small compared to the slot/base-cycle ILPs. As a result, the computational complexity of the x-offset calculation is negligible compared to the slot/base-cycle calculation.

3.1.4 Multi-stage ILP: ILP- based Heuristic Solution

The single-stage ILP has a limited scalability for FlexRay 3.0 where messages of all ECUs are scheduled concurrently. For instance, determining a schedule for 130 messages already requires more than 24 hours. However, if the scheduling problem is partitioned as it is the case for FlexRay 2.1[5], the scalability is only limited by the number of messages sent by each single ECU instead. This makes the single-stage ILP applicable for FlexRay 2.1 scheduling of a full state-of-the-art in-vehicle network since the number of messages sent by each ECU is usually much smaller. As a remedy, this section presents a multi-stage ILP which partitions the scheduling problem, i.e., it determines a message to slot assignment for each ECU individually, using the single-stage ILP. A second ILP allows to combine slots to a single slot, following the FlexRay 3.0 specification, see Figure 3.4 (page 50).

Figure 3.7 illustrates the optimization flow of the multi-stage ILP. In the first stage, the single-stage ILP is applied to each ECU e individually. It implicitly determines a frame $f \in F$

[5] For FlexRay 2.1 the messages sent by each ECU are scheduled individually since slot sharing is not permitted.

for each used slot. A frame $f \in F$ contains the messages $m \in M_f$. M_f is defined for a slot sl as follows:

$$m \in M_f \iff \exists b \in \{0, .., \rho_m - 1\} : \bar{\mathbf{z}}_{(m,sl,b)} = 1$$

Thus, if a message m is scheduled in slot sl, it is also part of the message set M_f of frame f contained in sl[6]. A frame uses the cycles C_f which is defined as follows:

$$c \in C_f \iff \exists m \in M_f, n \cdot \rho_m + b = c : \bar{\mathbf{z}}_{(m,sl,b)} = 1$$

Therefore, all cycles used by the messages in slot sl are also used by frame f. In the second stage, an ILP integrates the individually generated subsystem schedules in a global schedule. Hence, it assigns frames f using the cycles C_f to slots, determining the final slot and base-cycle for each of the frames. The results obtained by this multi-stage approach might not be optimal anymore, but the scalability is significantly increased. The ILP formulation is based on the following constants:

- $sl \in Sl$ - denotes index for slot.
- $m \in M$ - message, where M_f indicates all messages sent in the frame f.
- C_f - set of cycles used by frame f, hence, cycles in which at least one message is sent.
- ρ_f - smallest repetition of all messages scheduled in f, $\rho_f = \underset{m \in M_f}{\operatorname{argmin}}(\rho_m)$, determining the largest possible base-cycle for f.

The ILP uses the following variables:

- $\bar{\mathbf{y}}_{sl} \in \{0, 1\}$ - binary variable indicating if a slot sl is allocated.
- $\bar{\mathbf{u}}_{(f,sl,b)} \in \{0, 1\}$ - binary variable indicating if f is scheduled in slot sl with base-cycle b.

The integration ILP is formulated as:

$$\min \sum_{sl \in Sl} \bar{\mathbf{y}}_{sl} \tag{3.13}$$

$\forall f \in F$:

$$\sum_{sl \in Sl} \sum_{b=0}^{\rho_f - 1} \bar{\mathbf{u}}_{(f,sl,b)} = 1 \tag{3.14}$$

[6]The variable $\bar{\mathbf{z}}_{(m,sl,b)}$ indicates that message m is scheduled in slot sl with a base-cycle of b as introduced in the previous subsection.

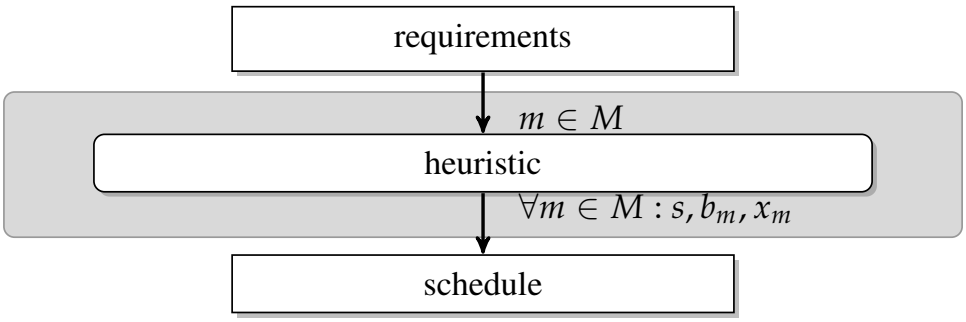


Figure 3.8: *Schedule synthesis with the greedy heuristic.*

$\forall sl \in Sl, c = \{0, ..., c_{\text{fr}} - 1\}$:

$$\sum_{f \in F} \sum_{\substack{b \in \{0,..,\rho_f - 1\} \\ (c-b) \bmod c_{\text{fr}} \in C_f}} \bar{\mathbf{u}}_{(f,sl,b)} \leq 1 \tag{3.15}$$

$\forall sl \in Sl, f \in F, b = \{0, ..., \rho_f - 1\}$:

$$\bar{\mathbf{y}}_{sl} - \bar{\mathbf{u}}_{(f,sl,b)} \geq 0 \tag{3.16}$$

The objective function (3.13) minimizes the number of allocated slots. The upper bound is the number of slots required if slots are not shared. Constraint (3.14) ensures that each f is only placed once. It also implies that b must be smaller than ρ_f. Furthermore, each cycle in a slot cannot be used by more than one frame as stated in Constraint (3.15). As the used cycles in C_f do not necessarily have a constant repetition, all base-cycles which would lead to a utilization of cycle c are determined. Constraint (3.16) ensures that the frame f can only be scheduled in slots that have been allocated.

In the automotive industry, legacy subsystems commonly have to be integrated in the architecture. Configurations for these subsystems have already been intensively tested and certified for previous product generations. The multi-stage approach allows to integrate these subsystem schedules in a global system, clearly minimizing the efforts for testing and certification.

3.1.5 Greedy Heuristic

This section first presents a greedy approach for schedule synthesis, before an optimized order for the greedy approach as well as GA and SA approaches are proposed.

3.1.5.1 Schedule Synthesis

In the following, a schedule synthesis based on a greedy approach is presented. For FlexRay 3.0, this approach is applied to the entire set of messages. For FlexRay 2.1, slots cannot be shared

Algorithm 3.1: Greedy heuristic for FlexRay scheduling.

Input: set of messages M
Output: slot sl_m and base-cycle b_m assignment for each message $m \in M$

```
1  Sl = ∅
2  σ(M) = defineOrder(M)    // sorted message list
3  for m ∈ σ(M) do
4      for sl ∈ Sl do
5          C_(sl,ř) = availableCycles(sl, ř(m))
6          if place(m, sl, C_(sl,ř)) then
7              continue with next m
8          end
9      end
10     create new sl and add it to Sl
11     C_(sl,ř) = {0, 1, ..., c_fr − 1}
12     place(m, sl, C_(sl,ř))
13 end
```

by ECUs and, thus, the approach is applied to the message set of each ECU separately and the slots are finally combined to a schedule. In contrast to the ILP approaches, the greedy approach assigns the slot sl, basecycle b_m and offset x_m to a message m in a single iteration, see Figure 3.8. With $C_{(sl,\check{r})}$ denoting the set of cycles in slot sl in which the ECU $\check{r}$ might send its messages, the greedy heuristic is outlined in Algorithm 3.1. It iteratively fills the slots with messages and allocates new slots if necessary.

The algorithm starts with an empty set of slots Sl. After the messages M have been put in order (line 2), the heuristic tries to place each message m in one of the existing slots sl (line 3-9). If a message cannot be placed in any existing slot in Sl, a new slot sl is allocated and the message is placed in this new slot (line 10-12). Before a message is placed in sl, the function $\text{availableCycles}(sl, \check{r}(m))$ determines the set of cycles $C_{(sl,\check{r})}$ in which the sender of the current message $\check{r}(m)$ is allowed to schedule its messages (line 5). This set of cycles $C_{(sl,\check{r})}$ might be sparse in case another ECU already scheduled a message in the same slot. In a next step, the function $\text{place}(m, sl, C_{(sl,\check{r})})$ tries to place the message m into the slot sl by considering the available cycles and determining a valid base-cycle b_m and offset x_m. Here, a message m can be placed in the slot sl if (1) the base-cycle b_m is smaller than the message repetition ρ_m and (2) m does not intersect with other messages for all its repetitions. The function $\text{place}(m, sl, C_{(sl,\check{r})})$ places messages in lower cycle numbers first and fills them from the left to achieve a dense scheduling. If a suitable b_m and offset x_m can be found, $\text{place}(m, sl, C_{(sl,\check{r})})$ returns *true* and the algorithm continues with the next message. In case the message cannot be scheduled in any existing slot, a new slot is allocated in which the message is scheduled. The proposed heuristic is a greedy algorithm and the runtime depends on the number of messages

as well as the number of assigned slots. The complexity is therefore defined as $\mathcal{O}(|M| \cdot |Sl|)$, being polynomial in the number of messages as $\mathcal{O}(|M| \cdot |Sl|) \leq \mathcal{O}(|M|^2)$.

3.1.5.2 Determine Message Order

The order of messages $\sigma(M)$ strongly affects the results obtained by the heuristic. The following order is proposed: (1) Messages are arranged by ascending repetition, (2) messages with the same repetition are arranged by their message size in descending order. The greedy heuristic benefits from (1), as messages with the same repetition are scheduled in one group and can therefore be easily combined in one slot. At the same time, messages with a high repetition rate are placed later and used to fill unused cycles in a slot as they occupy less cycles ($\frac{c_{\text{fr}}}{\rho_m}$) and are therefore more flexible in their placement. Similarly, (2) ensures that large messages are placed first while smaller messages are used to fill remaining payload in the slot.

In the following, the order is introduced formally. Given a set of messages $\{m_0, m_1, .., m_n\} \in M$, the message order is defined by the permutation $\sigma : \mathbb{N}_0 \rightarrow \mathbb{N}_0$. Arranging the message set by the repetition ρ_m is defined as:
$\forall m \in M, i, j \in \mathbb{N}_0, i \neq j, r_{m_{\sigma(i)}} \leq r_{m_{\sigma(j)}}$:

$$0 \leq i \leq j \leq n \tag{3.17}$$

The resulting list is then defined as $\sigma(M) = \{m_{\sigma(0)}, m_{\sigma(1)}, .., m_{\sigma(n)}\}$. As two messages might have equal repetitions, messages with a larger size l_m are additionally ordered to the front:
$\forall m \in \sigma(M), i, j \in \mathbb{N}_0, i \neq j, r_{m_{\sigma(i)}} = r_{m_{\sigma(j)}}, l_{m_{\sigma(i)}} \geq l_{m_{\sigma(j)}}$:

$$0 \leq i \leq j \leq n \tag{3.18}$$

The ordered message list $\sigma(M)$ improves the results obtained by the greedy heuristic compared to an unsorted list as shown in the results.

3.1.5.3 Genetic Algorithm-optimized Order

In the following, an efficient Genetic Algorithm (GA) [Mic96] approach is presented for FlexRay scheduling. The main procedure of an GA is based on reproduction and selection that are performed alternately in an iterative process to optimize a given objective. Reproduction creates new individuals from the current population, using *mutation* and *crossover* operators. For the experimental results, a population size of 100 is used, generating 25 offspring individuals in each generation. The task of selection is to remove the worst individuals in terms of their fitness function to ensure a convergence of the algorithm towards the optimal solutions. Here, elitism selection is used that always removes the worst 25 individuals in each generation. These parameters are default values of the used optimization framework [LGRT11] and might be adapted to further improve the results which, however, is beyond the scope of this thesis. In our experiments, we use 200 generations and, thus, the optimization is stopped after

5000 evaluated solutions. Generally, a longer runtime of meta-heuristics improves the quality of results.

One important factor of a GA optimization is the problem encoding. While it would be possible to encode the mapping of each message to a slot, this would result in many infeasible solutions as particularly for larger problems obtaining overfull slots is highly probable. Instead, we use a decoding-based approach such that each individual is feasible: The genetic representation of an individual in this case is a permutation of messages $\sigma(M)$ that is always decoded to a feasible scheduling, using the greedy heuristic (Algorithm 3.1). In this case, the initial population is determined by a random order of messages $\sigma(M)$ while the fitness function is the number of required slots of the currently determined schedule. As a result, the runtime for the evaluation of a single individual is dominated by Algorithm 3.1 which is polynomial.

For the crossover and mutation operator, we rely on the standard operators in [LGRT11]. Here, the crossover takes a sublist of the first permutation from the beginning to a random cut point and fills the remaining elements from the second permutation. The mutation is applied with a probability of $1/|M|$ and either moves a single element, swaps two elements, or reverts a random sublist.

3.1.5.4 Simulated Annealing-optimized Order

We further apply Simulated Annealing (SA) [KGV83] which is an optimization algorithm inspired by the annealing process in metallurgy. The algorithm varies a single solution by a neighborhood operator. The common neighborhood operator for binary problems is a bit flip or the sampling from a normal distribution for real-valued problems. Corresponding to the GA approach, a permutation of messages is used to represent a solution to ensure the feasibility of the solution by using the greedy heuristic from Algorithm 3.1 as decoder. The neighbor is determined by the neighbor operator for permutations in [LGRT11]: Either a single element is moved, two elements are swapped, or a sublist is reverted.

SA improves a single solution iteratively. Here, we accept and continue with an order $\sigma(M)'$, which is the neighbor of $\sigma(M)$, if $\nu(\sigma(M)') < \nu(\sigma(M))$ where $\nu(\cdot)$ is the fitness function that uses the heuristic in Algorithm 3.1 to determine the number of required slots. In case the neighbor solution does not improve the number of slots, it is accepted with a probability of

$$e^{\frac{\nu(\sigma(M))-\nu(\sigma(M)')}{T_i}} \tag{3.19}$$

where T_i is the temperature at iteration i. As cooling function,

$$T_i = T_0 \cdot \frac{n-i}{n} \tag{3.20}$$

is used with an initial temperature of $T_0 = 1$ and the number of iterations $n = 5000$. A high number of iterations generally improves the solutions but also increases the runtime linearly.

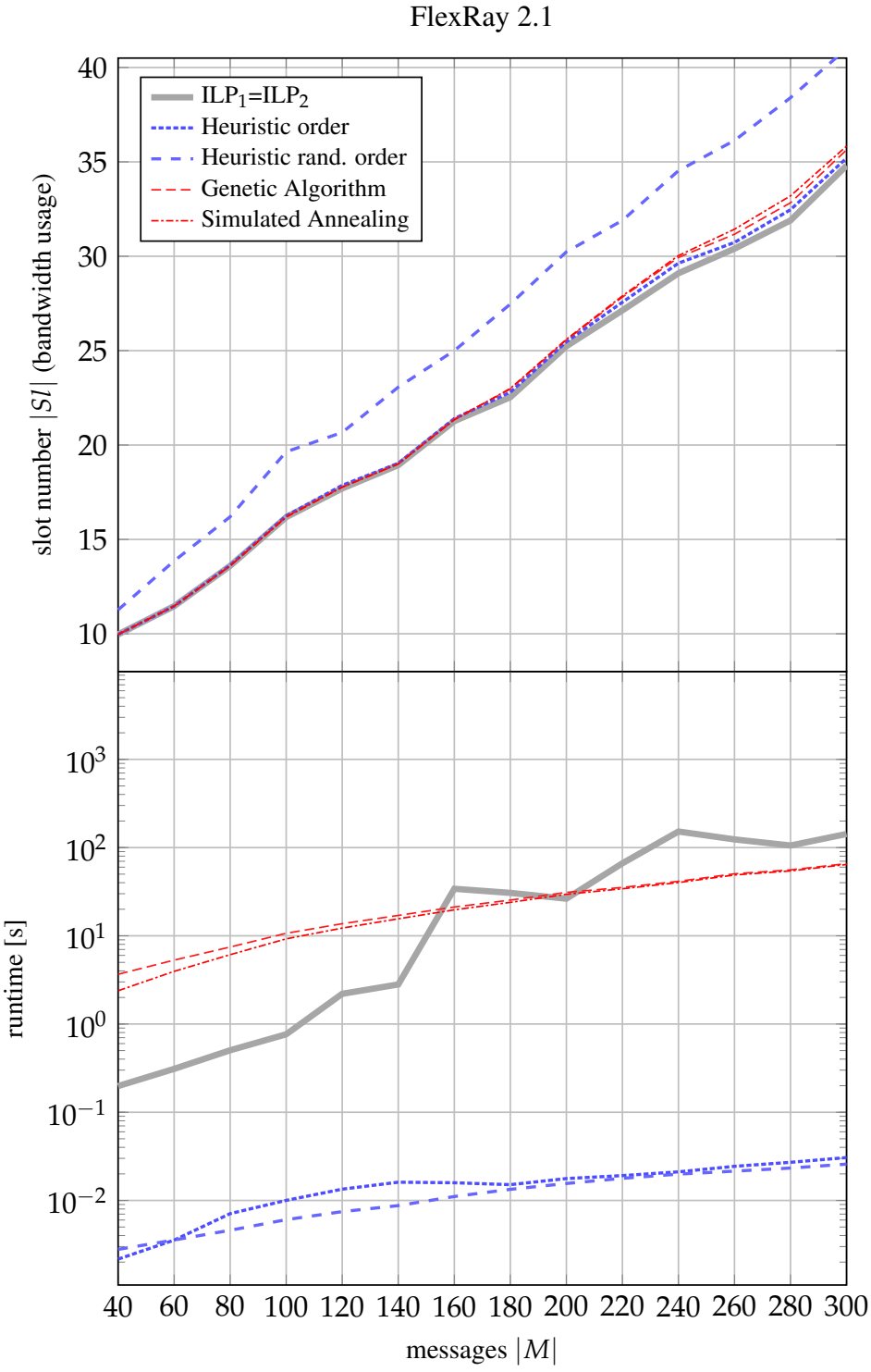


Figure 3.9: *Analysis of bandwidth utilization and runtime for 420 synthetic test cases for FlexRay 2.1. To improve legibility, the test cases are grouped by their message number and the average value displayed.*

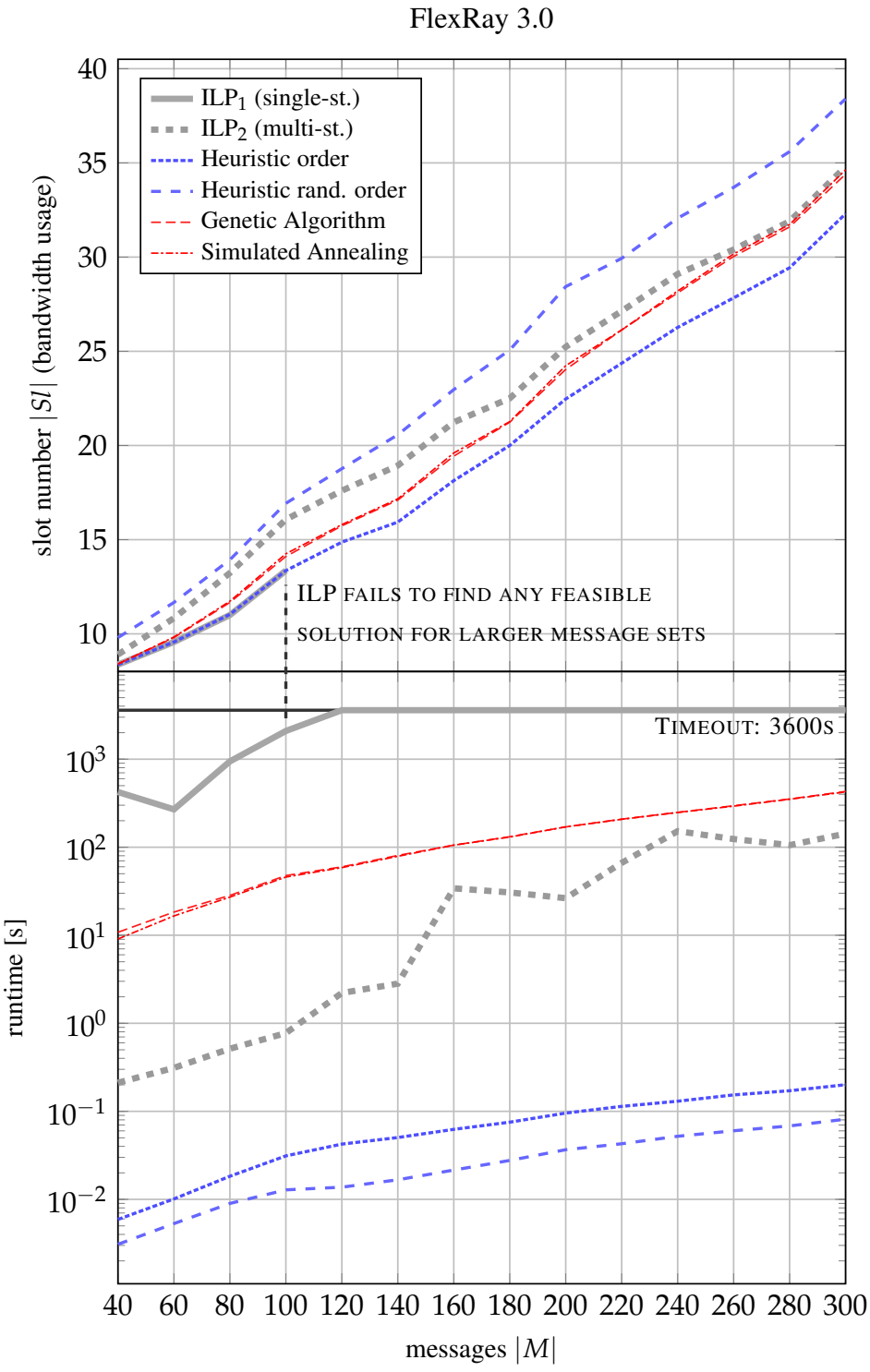


Figure 3.10: *Analysis of bandwidth utilization and runtime for* 420 *synthetic test cases for FlexRay 3.0.*

3.1.6 Experimental Results

To evaluate the schedule synthesis techniques, an extensive performance analysis and a large automotive case study is presented. All optimizations have been carried out on an Intel Xeon 3.2 GHz Quad Core with 12GB RAM. CPLEX in version 12.6 [ILO] was used as ILP solver for solving the ILP formulations and Opt4J 3.1.2 [LGRT11] for solving the GA and SA. As a measure for the quality of the obtained schedules, the number of slots required to schedule a set of messages is used. One additional major criterion is the runtime of the method since in particular methods based on ILP formulations might not scale well. Note that the schedule is obtained at design time such that runtimes of several minutes and even hours are still acceptable. As an upper bound for the number of slots for the single-stage ILP formulation the result obtained by the greedy heuristic is used. The times stated include the generation of the subsystem schedules and message offsets. In the following, first an extensive performance analysis is presented. Second, an automotive case study representing a full in-vehicle network is discussed.

3.1.6.1 Performance Analysis

In the following, an extensive case study consisting of 420 grouped synthetic test cases is presented. The size of the test cases ranges from 40 to 300 messages sent by 8 ECUs. The message size and period distribution is similar to message sets used in the automotive industry. For FlexRay 2.1, the schedule obtained by the multi-stage ILP is identical with the single-stage ILP.

Figure 3.9 shows the results obtained for this case study with FlexRay 2.1, while Figure 3.10 illustrates the results for FlexRay 3.0. The number of FlexRay cycles is $c_{\text{fr}} = 64$ for both FlexRay 2.1 and version 3.0. The results show that all approaches enable a better bandwidth utilization with FlexRay 3.0. At the same time, the runtime is increased due to the increased complexity. For FlexRay 3.0 the single-stage ILP is unable to terminate its calculation before reaching a time-out of one hour for test cases with more than 100 messages. If the greedy heuristic is applied to an ordered message set, it is able to determine close to optimal results for both FlexRay versions. The GA and SA approaches are also able to determine close to optimal results for small message sets, but with an increasing problem size they are outperformed by the heuristic with an ordered message set. For FlexRay 2.1, for 6.0 % and 4.0 % of the test cases, respectively, GA and SA find a better solution than the heuristic which finds a better solution for 13.6 % of the test cases. For FlexRay 3.0, the heuristic finds a better solution for 83.5 % of the test cases while GA and SA do not outperform the heuristic for any test case. Increasing the number of iterations or fitness function evaluations, respectively, for GA and SA or improving their parameterization would reduce this gap at the cost of additional runtime or configuration. For FlexRay 3.0, the multi-stage ILP is able to reduce the bandwidth utilization for several test-cases compared to FlexRay 2.1 scheduling. Finally, if the greedy heuristic is applied to a randomly-ordered message set, it is clearly outperformed by all other approaches. The greedy heuristics determine all their solutions in less than a second. Note that the runtime of the metaheuristics scales linearly in the number of fitness function evaluations while each

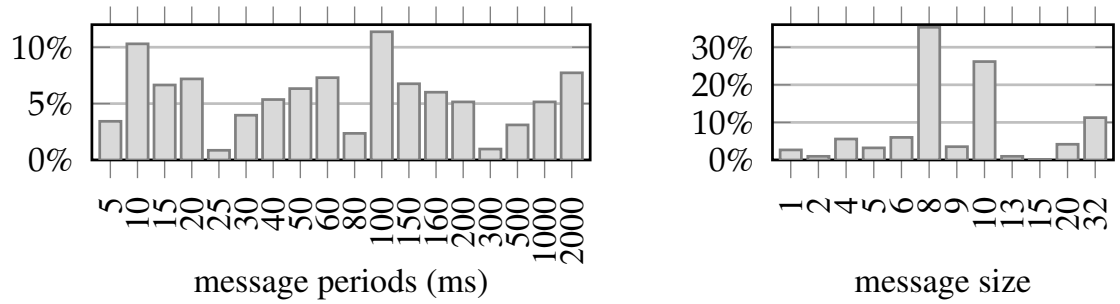


Figure 3.11: *Distribution of message periods and sizes for automotive case study.*

of these requires the decoding with the greedy heuristic and, thus, for 5000 evaluations their runtime is about three orders of magnitude higher. The runtime of the multi-stage ILP for FlexRay 3.0 strongly depends on the runtime of the first stage, determining a schedule for each ECU individually. Here, the runtime increase depends less on the overall number of messages, but rather the number of messages sent by each ECU.

Please note, the runtime in general increases with the number of messages. However, the test cases differ in their complexity. Single test-cases might therefore have an increased runtime compared to test cases with a larger message number. For instance, the deviations at 160 and 240 for the multi-stage ILP are resulting from single test cases with an increased complexity.

3.1.6.2 In-vehicle Network Case Study

This section presents a case study modeling an entire state-of-the-art in-vehicle network connected by a FlexRay bus. The network consists of 32 ECUs, sending 932 messages of different sizes and periods as illustrated in Figure 3.11. The parameters of the FlexRay bus were predefined such that the static segment consists of 62 slots with a slot payload of 42 bytes. As AUTOSAR requires that 1 byte of the payload is reserved for update bits, only $l_{\text{fr}} = 41$ bytes can be used for scheduling. The duration of a communication cycle was set to $h_{\text{fr}} = 5\,\text{ms}$.

The results in Table 3.2 show that with FlexRay 2.1 it is not possible to schedule all messages within 62 slots. For FlexRay 3.0, the greedy heuristic based on a ordered message set is able to schedule the messages with the given FlexRay parameters. If, in addition, the cycle number is selected as 60, the GA and SA algorithms are also able to schedule all messages. 60 cycles lead to a clear reduction of bandwidth utilization for all approaches. Here, the number of slots might be reduced by up to 10 %, depending on the applied approach. The results furthermore show that the ILP approach is unable to find a solution within 24 hours. The multi-stage ILP shows its ability to efficiently integrate independent schedules but is unsuitable for the current case study.

Table 3.2: *Results of automotive case study.*

Method	**60 cycles**		**64 cycles**	
	runtime	slots	runtime	slots
FlexRay 2.1				
$ILP_1 = ILP_2$		-	3.8s	$76^!$
Greedy order		-	0.039s	$76^!$
Genetic Algorithm		-	172.2s	$76^!$
Simulated Annealing		-	173.4s	$76^!$
FlexRay 3.0				
ILP_1 (single-st.)	24h[1]	-	24h[1]	-
ILP_2 (multi-st.)	88.0s	$63^!/70$[2]	448.5s	$69^!/76$[2]
Greedy order	0.76s	54	1.01s	60
Genetic Algorithm	2198.1s	57	3167.9s	$63^!$
Simulated Annealing	2218.4s	57	3201.6s	$63^!$

$^!$ exceeds number of slots in static segment

[1] timeout

[2] result without applying integration ILP

3.1.7 Summary

This section proposed a generic framework to schedule the static segment of the FlexRay bus using asynchronous communication. The presented results showed the clear potential of FlexRay 3.0 to improve the bandwidth usage compared to version 2.1, making optimal approaches for FlexRay 3.0 necessary. Previous work largely addressed FlexRay 2.1 and does not extend to version 3.0 while existing approaches for 3.0 are non-optimal in several aspects. By contrast, the scheduling approaches presented here support all features of version 3.0 while being backwards compatible. Several scheduling approaches were presented. (1) A single-stage ILP approach determining an optimal solution. (2) A multi-stage ILP, integrating previously generated subsystem schedules into a global schedule. (3) A greedy heuristic obtaining good results with a minimal runtime. (4) A GA and a SA approach to evaluated the proposed approaches.

An extended performance analysis and a realistic case study were presented. The results showed that the proposed algorithms are capable of finding feasible solutions for both FlexRay 2.1 and FlexRay 3.0. They also reflected the benefits of the new FlexRay 3.0 features to reduce the bandwidth requirements. At the same time, it was shown that for FlexRay 3.0, the ILP approach does not scale well and becomes intractable for large sets of messages. However, for small size problems where runtimes of up to several days are acceptable, it provides an optimal solution. The greedy heuristic generates highly competitive results in a short time

if the messages are arranged by their repetition and size before scheduling. It might therefore be applied for large message sets where the ILP approach becomes intractable. The GA and SA approaches were only able to obtain better results than the heuristic for small test cases and FlexRay 2.1. For large message sets and, in particular FlexRay 3.0, the heuristic clearly outperformed the metaheuristics in terms of obtained results and runtime. Furthermore, the results showed that the multi-stage ILP is suitable for optimizing schedules in an incremental design approach. It might be applied to convert existing legacy FlexRay 2.1 schedules to 3.0, or to integrate existing subsystems into an architecture if reduced testing and integration efforts are more important than optimal bandwidth usage. The large case study showed the necessity for FlexRay 3.0 where the novel features help to obtain a feasible schedule. Here, in particular setting the number of cycles to 60 allowed a better bandwidth utilization.

The multi-stage ILP presented in this chapter is based on a schedule integration approach that integrates individually generated FlexRay schedules into a global schedule for asynchronous scheduling. In Chapter 4, schedule integration is further investigated in the context of synchronous time-triggered systems to enable a modular schedule synthesis.

The focus in this section was on asynchronous scheduling. The following section will present an extension of the single-stage ILP to support a synchronous scheduling which also considers the task scheduling on ECUs.

3.2 Synchronous Time-Triggered Scheduling

With the introduction of an increasing number of advanced driver assistance functionality such as ACC, the timing requirements of automotive architectures are getting stricter. Synchronous time-triggered scheduling helps to increase the predictability of the system and give clear timing guarantees as all maximum end-to-end delays are fixed. The previous section has introduced a schedule synthesis for asynchronous message scheduling on the FlexRay bus. In the following, this approach is extended with a concurrent task scheduling and a support for Automotive Ethernet. The resulting ILP formulation is applied to generate subsystem schedules for the schedule integration approach presented in Chapter 4 and to evaluate our approaches in the following chapters.

Problem description. This section addresses the problem of defining a global schedule for all tasks and messages for ECUs communicating with one or more FlexRay/ Ethernet buses. As discussed in detail in Chapter 2, we assume a system specification which already defines a mapping of tasks to ECUs as it is commonly the case in the automotive domain. The schedule synthesis then addresses the problem of assigning start-times for all tasks and release-times for messages such that each resource is utilized exclusively (cf. Requirement 2.2.1), all precedence constraints are satisfied (cf. Requirement 2.2.2), and all end-to-end timing delays are met (cf. Requirement 2.2.3). For FlexRay scheduling, in addition a message to slot packing is defined.

Outline. In the following, Section 3.2.1 first gives an overview over related work. Section 3.2.2 introduces the basic formulation for an ILP-based schedule synthesis. Section 3.2.3 discusses an extension to support FlexRay scheduling. Finally, Section 3.2.4 extends these equations to support Automotive Ethernet.

3.2.1 Related work

The discussion of related work in the introduction has already given an overview of work done in the area of holistic time-triggered scheduling. Existing heuristic and meta-heuristic approaches generally scale well for large problems [Obe11]. However, for strongly constrained problems, e.g., for tight end-to-end delays and a high utilization, the approaches might be unable to find a feasible solution and back-tracking is required [HLW+14]. In addition, the results obtained by heuristic approaches might be suboptimal. As a consequence, we rely on mathematical techniques to implement a holistic schedule synthesis which provides a baseline for the approaches presented in the following chapters. Two recent approaches addressing holistic scheduling for a system communicating with a FlexRay bus have been proposed in [ZDGSV11, LSGC12]. Both papers apply an ILP to determine a system schedule. The ILP formulation presented in the following is based on these approaches, adding a support for Automotive Ethernet.

3.2.2 Schedule Synthesis

In the following, a schedule synthesis approach for holistic scheduling of a synchronous time-triggered system is presented. The schedule synthesis is based on the following constants.

- p - process referring to both tasks and messages. Defined in detail in Definition 2.1.1.
- h_p - process period defining time interval after which p is executed again.
- τ_p - execution-time for a task or transmission time for a message.
- $a \in A_d$ - defines an application with its task graph $G_a = (P_a, E_a)$. Described in detail in Definition 2.1.2.
- d - specification describing a task graph $G_d = (P_d, E_d)$. Described in detail in Definition 2.1.4.
- $r(p) : P \to R$ - returns the predefined process mapping to a resource. Defined in detail in Definition 2.1.3.
- $h(p, \tilde{p}) = \text{lcm}(h_p, h_{\tilde{p}}) : P \to \mathbb{R}$ - returns the hyper period of the process periods of p and $\tilde{p}$. Here, $\text{lcm}(\cdot)$ defines the least common multiple. Defines period after which the schedule for p and $\tilde{p}$ repeats.

- $e = (p, \tilde{p}) \in E$ - data-dependency of $\tilde{p}$ from p, defining the process precedence.
- θ_a - deadline of application a. Defines maximum end-to-end delay from all source to all sink processes.
- ϕ - path or subgraph from a source to a sink task, e.g., $\phi = \{p_1, p_2, .., p_n\}$. The processes must be pairwise connected with an edge e. Defined in detail in Definition 2.2.4.
- $\Phi(E_a) : \mathcal{E} \rightarrow \{\Phi\}$ - returns all paths ϕ of application a. Applies a *Depth-first search* algorithm to determine paths [Eve11].

In addition, the schedule synthesis approach is based on the following variables.

- $\mathbf{s}_p \in \mathbb{R}$ - variable for the start-time of p.
- $\mathbf{f}_p \in \mathbb{R}$ - variable for finish-time of p.
- $\mathbf{w}_{(p,\tilde{p})} \in \mathbb{R}$ - variable for waiting-time between two data-dependent processes $p, \tilde{p}$. The waiting-time is defined as the delay between the finish-time of p and start-time of $\tilde{p}$.
- $\bar{\mathbf{q}}_{\mathbf{f}_p} \in \{0, 1\}$ - binary variable indicating if $\mathbf{f}_p$ exceeds the period of p.
- $\bar{\mathbf{q}}_{\mathbf{w}_{(p,\tilde{p})}} \in \{0, 1\}$ - binary variable indicating if $\mathbf{w}_{(p,\tilde{p})}$ exceeds the period of p and $\tilde{p}$.
- $\bar{\mathbf{x}}_{(p,\tilde{p},i,j)} \in \{0, 1\}$ - binary variable indicating if p is placed before $\tilde{p}$ for the indexes i and j.

The goal of the schedule synthesis is to determine a synchronous time-triggered schedule as illustrated in Figure 3.12. The figure also illustrates the basic variables determined during schedule synthesis. The ILP approach for time-triggered scheduling is then defined as follows.
$\forall p \in P_d$:

$$0 \leq \mathbf{s}_p < h_p \tag{3.21}$$

$$0 \leq \mathbf{f}_p < h_p \tag{3.22}$$

$\forall a \in A_d, e \in E_a, (p, \tilde{p}) = e$:

$$0 \leq \mathbf{w}_{(p,\tilde{p})} < h_p \tag{3.23}$$

$\forall p \in P_d$:

$$\mathbf{f}_p = \mathbf{s}_p + \tau_p - \bar{\mathbf{q}}_{\mathbf{f}_p} \cdot h_p \tag{3.24}$$

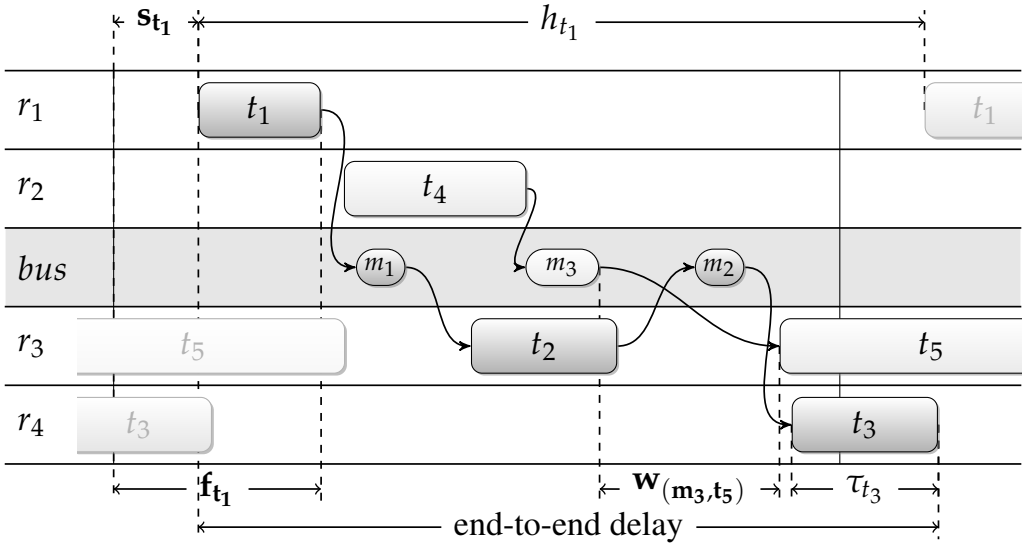


Figure 3.12: *Time-triggered schedule determined during a holistic schedule synthesis. The schedule defines a start-time* $\mathbf{s_p}$ *for each periodically executed task or message. Instead of considering task deadlines, we define a maximum end-to-end-delay for the application.*

$\forall a \in A_d, e \in E_a, (p, \tilde{p}) = e$:

$$\mathbf{w}_{(p,\tilde{p})} = \mathbf{s}_{\tilde{p}} - \mathbf{f}_p + \bar{\mathbf{q}}_{\mathbf{w}_{(p,\tilde{p})}} \cdot h_p \tag{3.25}$$

$\forall p, \tilde{p} \in P_d, p \neq \tilde{p}, r(p) = r(\tilde{p}), i = \left\{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_p} - 1\right\}, j = \left\{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\right\}$:

$$\mathbf{s}_p + \tau_p + i \cdot h_p \leq \mathbf{s}_{\tilde{p}} + j \cdot h_{\tilde{p}} + 2 \cdot h(p, \tilde{p}) \cdot (1 - \bar{\mathbf{x}}_{(p,\tilde{p},i,j)}) \tag{3.26}$$

$$\mathbf{s}_{\tilde{p}} + \tau_{\tilde{p}} + j \cdot h_{\tilde{p}} \leq \mathbf{s}_p + i \cdot h_p + 2 \cdot h(p, \tilde{p}) \cdot \bar{\mathbf{x}}_{(p,\tilde{p},i,j)} \tag{3.27}$$

$\forall a \in A_d, \phi \in \Phi(E_a)$:

$$\theta_a \geq \sum_{p \in \phi} \tau_p + \sum_{(p,\tilde{p}) \in \phi} \mathbf{w}_{(p,\tilde{p})} \tag{3.28}$$

For the schedule synthesis a periodic scheduling model is applied, as defined in Definition 2.2.1. Constraints (3.21) and (3.22) therefore ensure that the process start-time as well as the finish-time are within the process period. As a consequence, also the waiting-time cannot exceed a process period, as stated in Constraint (3.23). Through the definition of a lower bound of 0, the constraint ensures that the precedence constraint defined in Requirement 2.2.2 is fulfilled. Based on these boundaries, the finish-time is defined in Constraint (3.24). The switching variable ensures that the finish-time does not exceed the process period. Similarly, the waiting-

time is defined in Constraint (3.25). The Constraints (3.26) and (3.27) ensure that two processes which are mapped to the same resource are executed consecutively. Hence, a process p either terminates its execution before a second process $\tilde{p}$ is started, or starts after the other process is terminated, ensuring that the exclusive utilization defined in Requirement 2.2.1 is satisfied. Finally, Constraint (3.28) ensures that the maximum end-to-end delay is not exceeded, as defined in Requirement 2.2.3.

These constraints determine a time-triggered schedule for tasks and messages of a system communicating with a bus that allows to selected slot start-times freely. In addition, an objective might be defined, as discussed in detail in Section 4.4.

3.2.3 FlexRay Scheduling

This section presents an extension for the holistic scheduling to integrate the asynchronous FlexRay schedule synthesis in Section 3.1. The FlexRay scheduling is based on the ILP determining an optimal solution in Section 3.1.3. The Constraints (3.4)- (3.8) (page 52) for asynchronous scheduling might then be integrated with the schedule synthesis for synchronous scheduling through the definition of message start-times (cf. Requirement 2.3.1). Thus, in addition to the constraints defined for asynchronous scheduling, we also define Constraints (3.21)-(3.25) and Constraint (3.28) for each FlexRay message. Here, the execution-time of a FlexRay message is defined as the duration of a static slot, $\tau_m = \tau_{\text{fr}}$, as a message must be available at the beginning of the slot it is transmitted in and it is only available once the whole slot was received. Hence, the delay introduced by the message transmission equals the slot duration. We assume that the first static slot starts at the time instant 0 (cf. Definition 2.3.1).

To integrate the asynchronous FlexRay constraints into a holistic schedule synthesis, an additional constraint is required. It is based on the binary variable $\bar{\mathbf{z}}_{(m,sl,b)}$ used for asynchronous FlexRay scheduling. $\bar{\mathbf{z}}_{(m,sl,b)}$ indicates if the FlexRay message m is sent in slot sl with a base-cycle of b. The start-time of a FlexRay message is then defined as:
$\forall m \in \{p | p \in P_d, \exists p : r(p) \in R_{\text{fr}}\}$:

$$\mathbf{s}_m = \sum_{sl \in Sl} \sum_{b=\{0,..,\rho_m\}} \bar{\mathbf{z}}_{(m,sl,b)} \cdot (sl \cdot \tau_{\text{fr}} + b \cdot h_{\text{fr}}) \tag{3.29}$$

The set R_{fr} represents all resources that are a FlexRay bus. Constraint (3.29) then defines that a FlexRay message might only be sent at the beginning of the slot it is assigned to. To determine the x-offset x_m for each message in the slot sl, we apply the ILP from Section 3.1.3.2.

FlexRay 2.1. For version 2.1 of FlexRay, slots can only be exclusively utilized by one ECU to send messages. For asynchronous scheduling, we therefore apply the schedule synthesis for each slot individually. As this is not feasible for a holistic scheduling, the equations for asynchronous scheduling need to be adapted for a holistic schedule synthesis. We therefore introduce an additional variable.

- $\bar{\mathbf{v}}_{(\check{r},sl)} \in \{0,1\}$ - binary variable indicating if slot sl is assigned to ECU $\check{r}$.

Constraints (3.7) and (3.8) (page 52) can then be adapted as follows.
$\forall sl \in Sl$:

$$\sum_{\check{r} \in \check{R}} \bar{\mathbf{v}}_{(\check{r},sl)} \leq 1 \tag{3.30}$$

$\forall sl \in Sl, \forall \check{r} \in \check{R}, m \in M(\check{r}), b = \{0,...,\rho_m - 1\}$:

$$\bar{\mathbf{v}}_{(\check{r},sl)} - \bar{\mathbf{z}}_{(m,sl,b)} \geq 0 \tag{3.31}$$

The repetition ρ_m represents the period of a message in FlexRay cycles: $\rho_m = \frac{h_m}{h_{\text{fr}}}$. The constraints ensure that only one resource is sending its messages in a slot defined by Requirement 2.3.2.

3.2.4 Scheduling for Automotive Ethernet

As explained in Section 2.3.3, for messages transmitted on the Ethernet bus we assume the same properties as for tasks mapped to a resource. Thus, applying the ILP formulation allows to determine a valid time-triggered schedule for Automotive Ethernet. However, as Automotive Ethernet is realized using a switched network and supports a full-duplex communication, this would not consider concurrent message transmission, overestimating the bandwidth requirements. Therefore, we introduce the following additional function:

- $\psi(p) : P \rightarrow \Psi$ - returns the routing for a message sent on the Ethernet bus, i.e., $\psi(p)$ returns all paths containing the data links that connect the sending resource to all receiving resources. For instance, $\psi(p) = \{(r, sw_1), (sw_1, sw_2), (sw_2, \tilde{r})\}$ describes the routing from resource r to $\tilde{r}$ via the switches sw_1 and sw_2.

The condition of Constraints (3.26) and (3.27) can then be adapted for Ethernet messages as follows.

$$\begin{aligned} &\forall p, \tilde{p} \in P_d, p \neq \tilde{p}, r(p) = r(\tilde{p}) \in R_{eth}, \psi(p) \cap \psi(\tilde{p}) \neq \emptyset, \\ &i = \left\{0,.., \frac{2 \cdot h(p,\tilde{p})}{h_p} - 1\right\}, j = \left\{0,.., \frac{2 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\right\} : \end{aligned} \tag{3.32}$$

Here, R_{eth} represents the set of all resources that are an Ethernet bus. Ethernet messages might be sent concurrently if the routings of the two messages do not share a common path. As the routings also define the direction of the link, this formulation also considers full-duplex communication. This formulation is based on the assumtion that the transmission time defined for an Ethernet message considers transmission delays introduced by the switches, as defined in Section 2.3.3.

3.2.5 Summary

This section presented an extension of the single-stage ILP introduced in Section 3.1 to support synchronous scheduling of a time-triggered system. A holistic scheduling was presented which considers the task scheduling on ECUs as well as the message scheduling on FlexRay and Automotive Ethernet communication buses. The approach is applicable for heterogeneous architectures consisting of multiple communication buses.

In the following, Chapter 4 presents a modular schedule synthesis approach to integrate subsystem schedules in a global schedule. The ILP approach presented in this section is then used to generate the subsystem schedules. In addition, the ILP approach is used to evaluate the schedule integration approach in Chapter 4 and the variant-aware schedule synthesis presented in Chapter 5.

4

Modular Schedule Synthesis for Heterogeneous Architectures

The previous chapter has introduced approaches for the schedule synthesis of time-triggered systems, both for asynchronous systems as well as synchronous systems. The focus there is on generating schedules from the scratch. In this chapter, a schedule synthesis approach is presented that allows to integrate independently created subsystem schedules into a global schedule. The problem is addressed for synchronous time-triggered scheduling.

The functionality in automotive E/E-architectures is largely developed by suppliers. While the software-based functionality today is still mainly implemented on ECUs, in the near future it is likely to be delivered to the manufacturer as AUTOSAR Software Components. The integration of all functionality into a seamless working system is a highly challenging task, as various constraints have to be taken into account for each application. These include resource constraints, data-dependency constraints and end-to-end timing delay constraints. Timing-analysis tools and defining resource requirements at an early design stage help to reduce the complexity of the integration process. However, they cannot substitute a systematic approach allowing a reliable and efficient integration of components. The demand for such integration methodologies will further rise with an increasing use of Over-The-Air programming (OTA) updates to fix software problems [WZ15], and the introduction of in-vehicle appstores.

This chapter addresses the problem of integrating individually generated subsystem schedules into a global schedule. We assume a time-triggered architecture as presented in Chapter 2, applying synchronous time-triggered scheduling. Figure 4.1 illustrates the specification of a small system on which three independently developed applications should be implemented. To

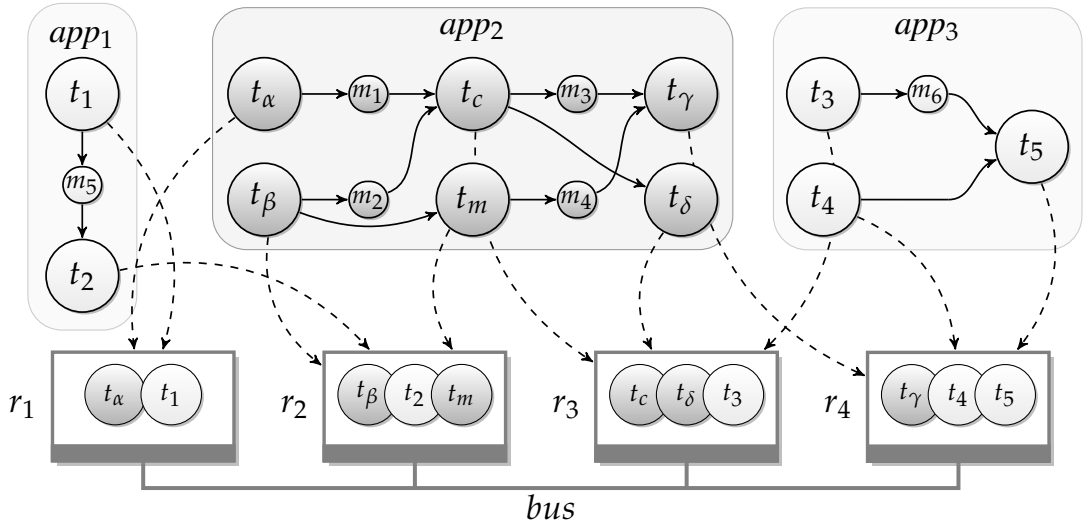


Figure 4.1: *System specification consisting of task graphs of three applications and their mappings to four ECUs connected by a communication bus.*

generate the system schedule, the manufacturer currently has to synthesize a schedule from the scratch or apply an incremental approach, trying to extend legacy schedules with new functionality. For time-triggered systems, the schedule is commonly defined at the specification phase [BÖ07]. Changes to the initially defined timing behavior or the introduction of additional tasks or messages to an application during the development then often require a time-consuming modification of the schedule to minimize the effects on other applications. As a remedy, this section introduces a *schedule integration* framework. It allows to integrate independently developed application/subsystem schedules into a global system schedule. This divide-and-conquer approach significantly reduces the integration complexity while the system becomes highly composable.

Methodology. Figure 4.2 illustrates the design process based on schedule integration for the applications specified in Figure 4.1. For each of the applications, a schedule synthesis is applied to determine a subsystem schedule, e.g., for testing and evaluation. To integrate the individual application schedules into a single global schedule, we apply a schedule integration approach. It ensures that the timing constraints of each subsystem are satisfied in the resulting global schedule, i.e., that the hard real-time requirements of all applications are met. The basis for the proposed schedule integration framework is formed by a time-triggered architecture, as it provides temporal composability [KB03, ATD$^+$09]. Hence, if a subsystem schedule can be integrated into a global schedule such that its general structure is not altered, all constraints, defined when the subsystem schedule was created, are still satisfied. At the same time, the complexity is reduced through this divide-and-conquer approach, as for integrating a subsystem schedule only a subset of constraints is required. For this example, we focused on single ap-

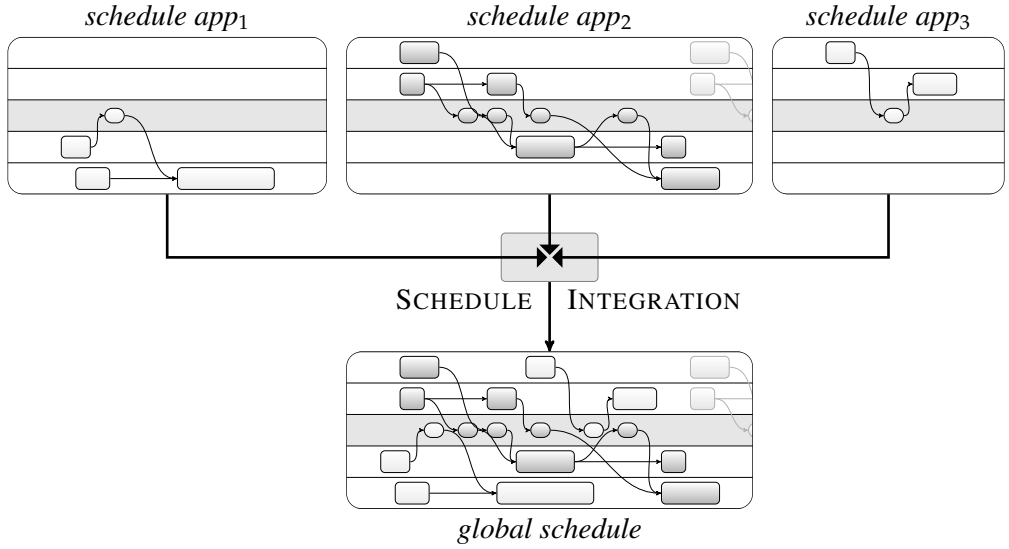


Figure 4.2: *Schematic illustration of schedule integration problem for the applications specified in Figure 4.1.*

plications. However, schedule integration might also be applied to clusters containing multiple applications such as the drivetrain or chassis domain.

Related work. In the area of *hierarchical scheduling* for *component-based* systems, the problem of integrating independent local schedulers into a global scheduling through assigning runtime budgets is studied [WT06, SL08]. The hierarchical scheduler provides isolation between the different components and allows to develop subsystems individually. However, hierarchical scheduling generally addresses single resources and existing approaches addressing distributed systems such as [KWL+12] have drawbacks. Another body of related work addresses the problem of *extensibility* of a time-triggered schedule [WJP+05, ZDGSV11]. The goal is to generate a system schedule such that extending the schedule with additional applications is facilitated. By contrast, schedule integration is concerned with integrating subsystem schedules in a common schedule rather than generating an extensible schedule from the scratch. Related to the problem of extensibility, several approaches have been presented for *incremental scheduling* [PEPP01, PEPP04]. Incremental scheduling addresses the problem of extending a legacy schedule with additional applications with the goal of minimizing the changes to the initial schedule. While incremental scheduling is concerned with extending a single schedule, schedule integration addresses the problem of integrating multiple schedules concurrently into a common schedule. Incremental scheduling approaches are therefore not directly applicable for schedule integration, while, in turn, schedule integration might be applied for incremental scheduling. A detailed discussion of related work is given in Section 4.1.

Contributions. This chapter proposes a framework for the integration of subsystem schedules in a global schedule. Compared to traditional approaches, schedule integration (1) allows to

significantly reduce the runtime of the schedule synthesis and (2) enables a highly modular system design, as subsystems can be easily added and updated.

We propose an SMT-based approach to combine all subsystem schedules to a single global schedule. As integrating individually generated subsystem schedules in a single schedule might not always be feasible, we also propose a conflict refinement to adapt conflicting schedules. During conflict refinement, all previously defined constraints, such as end-to-end delays are taken into account such that all timing requirements are fulfilled. For the conflict refinement, a partitioning heuristic identifies suitable subproblems which might be solved individually to further increase the scalability. The proposed schedule integration framework supports both Automotive Ethernet and FlexRay. As the integrability of different subsystem schedules depends on their structure, this chapter also investigates several metrics aiming at an improved integrability. The proposed schedule integration framework addresses synchronous time-triggered scheduling as it exploits the temporal composability of a time-triggered architecture. The focus here is not to obtain a globally optimal schedule, but rather to integrate locally optimized application configurations according to their specific requirements. We also propose a design flow based on a data-centric description which allows to integrate our framework into an AUTOSAR tool chain.

In summary, this chapter proposes a framework for scheduling modular time-triggered systems which is well-suited for solving large and complex scheduling problems.

Outline. In the following, Section 4.1 discusses related work. Section 4.2 introduces schedule integration formally. Section 4.3 presents the schedule integration approach supporting FlexRay and Automotive Ethernet. The section also presents our conflict refinement. Section 4.4 presents several metrics to improve the integrability of subsystem schedules. Section 4.5 analyzes our framework using three extensive case studies. Finally, Section 4.6 discusses how schedule integration might be integrated in an AUTOSAR-based design flow.

4.1 Related Work

The impact of software in automotive E/E-architectures is constantly growing, requiring new design paradigms to cope with the growing inherent complexity. The AUTOSAR partnership is one of the efforts undertaken to define an open and uniform software platform to improve the portability of software [AUT14]. Our schedule integration framework in combination with a data-centric description [PCFW05] might be integrated in the AUTOSAR tool chain to improve the composability of the architecture for time-triggered systems, as discussed in detail in Section 4.6. The focus lies on a modular schedule synthesis for synchronous time-triggered scheduling.

We would like to distinguish our framework from work which uses the term *modular scheduling* in a different context. The approaches presented in [Fea04, Fea06] address the problem of generating a schedule for the execution of a synchronous program in the context

of parallel computing. To improve the scalability of the schedule generation, the papers propose a modular scheduling, splitting a large program into several modules which might then be scheduled individually. This problem is very different from the schedule integration problem addressed here and is therefore not comparable with our work.

In the area of *hierarchical scheduling* for *component-based* systems, various research has been carried out on modular systems [SL03, WT06]. It addresses the problem of integrating independent local schedulers into a global schedule through assigning runtime budgets. This allows to apply different scheduling strategies like Earliest Deadline First (EDF) or Rate-Monotonic (RM) scheduling for individual schedulers. In contrast to this component-based scheduling which is often limited to single resources, automotive architectures implement distributed applications running on several networked resources. Hence, instead of defining deadlines for single tasks, a maximum end-to-end delay is defined. In [KWL$^+$12] hierarchical scheduling is applied to a distributed system by splitting the maximum global end-to-end delay into local deadlines which allows to estimate the required runtime budget for each single task. While this simplifies the problem, it also limits the obtainable schedules, e.g., in the achievable minimal end-to-end delay. In addition, splitting an application task-graph into multiple components which are scheduled by different local hierarchical schedulers makes changes to the system schedule difficult. By contrast, schedule integration allows to efficiently integrate and update application schedules.

Related to component-based design, in the area of *partition scheduling* various work has been done to integrate applications of different criticality levels in a single node [LKY$^+$00, TSP11]. Partition scheduling is often used in the context of Integrated Modular Avionics (IMA) [Pri92], a system partitioning approach which provides temporal and spatial isolation of applications. Partition scheduling aims at determining suitable partitions such that the timing constraints of all applications are satisfied. For instance, in [LKY$^+$00] a heuristic approach is presented to determine suitable partitions for an architecture consisting of distributed processors connected by a TDMA bus. The paper considers a preemptive, event-triggered execution of tasks within the partitions and defines the partitions as a time-triggered schedule for each node. In [TSP11] an approach to determine time-partitions for distributed applications is proposed. The paper applies an approach based on simulated annealing which considers both time- and event-triggered tasks. However, while partition scheduling is concerned with determining suitable system partitions, schedule integration is concerned with the integration of cluster schedules in a global schedule. Nevertheless, if partition scheduling is applied to determine suitable partition schedules for distributed applications, schedule integration provides an efficient way to integrate these distributed partitions into a global schedule. This would also allow to apply schedule integration for event-triggered scheduling.

Our framework is based on a fully time-triggered system, providing the basis for temporal composability [KB03, ATD$^+$09] which is an asset in this domain. Scheduling of time-triggered systems is a challenging task and various approaches have been proposed. The discussion of related work in the introduction has already given a basic overview of existing work for

time-triggered scheduling. In addition, a schedule synthesis approach using the model-checker *SAL* to determine a system schedule is presented in [VSE09]. An SMT-based approach to generate time-triggered schedules for *TTEthernet* is presented in [Ste10]. Finally, two ILP-based approaches are presented for the *FlexRay* bus in [ZDGSV11, LSGC12]. While [LSGC12] applies the schedule optimization on *task-level*, [ZDGSV11] addresses the problem on *job-level*. Most of these approaches perform well for small subsystem schedules, but do not scale well for larger systems. By contrast, the modular framework presented here is well-suited for solving large and complex scheduling problems. Schedule integration might therefore be applied to integrate the cluster schedules determined with one of the above mentioned approaches in a global system schedule.

Various work has been done in the area of extensibility of schedules. This body of work applies extensibility metrics during schedule synthesis to ease the integration of additional applications at a later stage. In this context, [SGC+11] proposes metrics for the message scheduling on the FlexRay bus. The paper considers the sustainability of a schedule, i.e., ensures that message deadlines will not be violated if the schedule is extended, and extensibility, i.e., the schedule can accommodate future messages. [ZDGSV11] presents a schedule synthesis approach for a holistic scheduling for a system communicating over a FlexRay bus. The paper proposes an extensibility metric minimizing the number of FlexRay slots and maximizing the slack usage. Similarly, the work in [WJP+05] proposes an extensibility metric for tasks and message scheduling in a system connected by a TDMA bus. Finally, in [PEPP04] two extensibility metrics are presented for non-preemptive scheduling in a time-triggered system, as addressed here. The metrics aim at (1) generating long connected idle time segments to accommodate tasks with a long execution-time, and at (2) providing periodic idle time segments to accommodate tasks with a small period. The metrics proposed in Section 4.4 also address the problem of extensibility and the proposed metrics share the objectives of maximizing the available slack and optimizing the use of idle time. In contrast to approaches addressing the extensibility of a system, schedule integration is concerned with the actual integration of individually developed schedules into a single global schedule.

Related to the problem of the extensibility of a schedule, various work has been done in the area of *incremental scheduling*. Incremental scheduling deals with the extension of an existing schedule with new functionality such that a minimal number of changes to the initial schedule is required. Extending the schedule is often combined with a second objective, optimizing the extensibility of the system. An approach for incremental scheduling of a system communicating with a TDMA bus is presented in [PEPP01]. The paper proposes a mapping heuristic, selecting suitable resources for tasks such that an existing legacy schedule is not affected when additional applications are integrated in the system. The approach also takes the extensibility of the generated schedule into account. In [PEPP04] an approach for incremental scheduling that addresses synchronous time-triggered systems is presented. The paper proposes a heuristic which identifies a minimal subset of processes which need to be shifted or mapped to a different resource for a TDMA-based task and message scheduling. Schedule integration might be ap-

plied for the problem of incremental scheduling, i.e., if additional constraints fixing the position of legacy schedules are defined. However, the aim of the proposed schedule integration framework is a modular schedule synthesis where multiple independently developed application and cluster schedules should be integrated in a global system schedule. Consequently, approaches for incremental scheduling are not directly applicable to the problem of schedule integration.

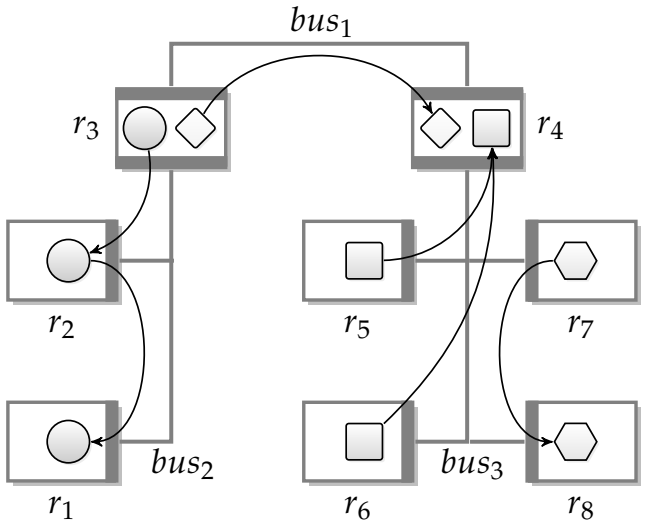
Finally, we have presented two approaches for schedule integration in [SLC13] and [SAW+14]. The approach in [SLC13] is concerned with schedule integration for the FlexRay bus. The focus lies on the message scheduling and the approach therefore differs from the schedule integration framework presented in this chapter. [SAW+14] forms the basis for the framework presented in this chapter and was extended to support both FlexRay and Automotive Ethernet.

4.2 Framework

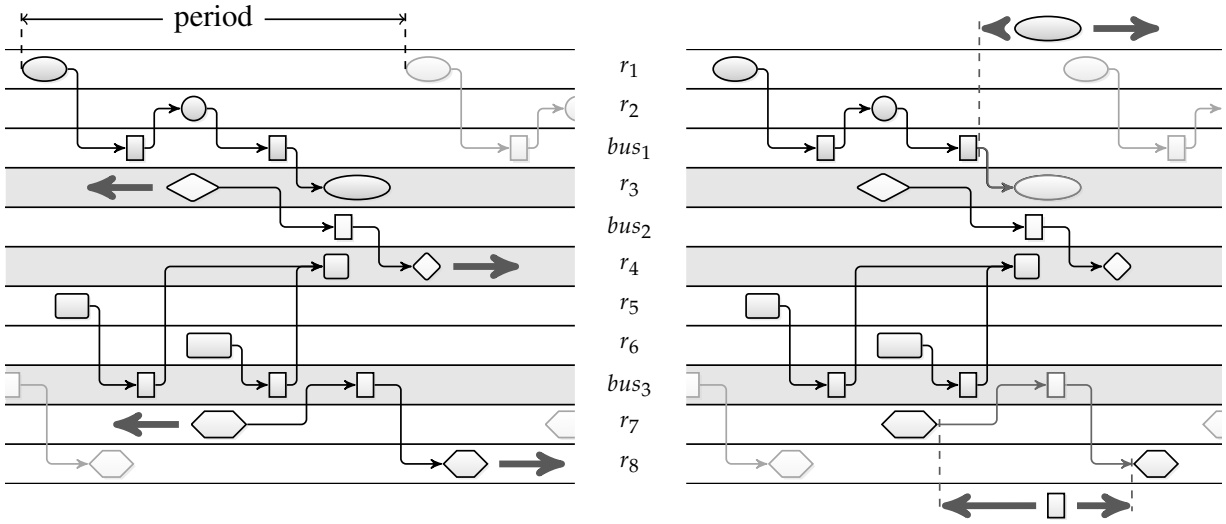
This section first introduces the basic principles of schedule integration before the proposed framework is presented.

4.2.1 Schedule Integration

The schedule integration addressed here is based on time-triggered synchronous scheduling as introduced in detail in Section 2.2.2. A schedule defines the exact timing of each of the tasks on the ECUs and the message transmission times on the buses. In state-of-the-art vehicles, various applications are implemented over several spatially distributed ECUs. Very often applications are developed independently or several applications are grouped to clusters based on their functionality with distinct requirements regarding performance, reliability, and safety. In this chapter, clusters are assumed to be predefined but there might exist also graph-based approaches to determine clusters of an entire system by using techniques like the approach presented in Section 5.3.3. While not all ECUs might be shared between clusters, through AUTOSAR an increasing number of applications and clusters will share ECUs. In the following, we will refer to all subsystems as clusters. An example of an architecture that shares resources between clusters is illustrated in Figure 4.3(a). For the sake of simplicity, each cluster is defined by a single application in this example. Defining a schedule for the complete system becomes a computationally intensive task, and mathematical methods like [LSGC12] and [ZZD+09] are not capable of delivering a feasible solution within a reasonable amount of time due to scalability issues. Moreover, it is common practice to develop each cluster separately and, thus, also to determine the schedules separately. In a final step, an integration of the subsystems becomes necessary such that for a large time-triggered system, a schedule integration has to be performed. To avoid conflicts in the resulting schedule, we propose two techniques as illustrated in Figure 4.3(b) and 4.3(c) which are described formally in the following.



(a) Domain-based architecture.



(b) Shifting of an entire cluster.

(c) Shifting of a single process.

Figure 4.3: *(a) The illustration shows an architecture consisting of two domains connected with the backbone bus_1. Four clusters (○, ◇, □, ⬡) share the resources r_3, r_4, and bus_3. (b) During schedule integration, the entire cluster schedules are shifted such that a feasible global schedule is obtained. (c) Additionally, cluster schedules might be modified by shifting single processes within the time frame defined by predecessor and successor processes.*

As defined in Chapter 2, one resource can only be utilized by one process p at a time instant. Hence, in many cases a direct schedule integration where cluster schedules are simply combined will lead to conflicts by violating Equation (2.2). As a remedy, a schedule integration approach is presented to merge existing schedules of subsystems that share resources. Thus, also the complexity of the scheduling is reduced by a divide-and-conquer approach. Cluster schedules are combined by shifting the whole schedule and process start-times, respectively. These two techniques increase the chance to find a feasible schedule for the entire system. The two techniques are described in the following for the general case.

One way to merge the schedules of two clusters d and $\tilde{d}$ is to shift the cluster schedules by a constant time which is denoted as $\mathbf{o}_d$ and $\mathbf{o}_{\tilde{d}}$, respectively. Shifting the schedule by a constant time does not influence the behavior of the subsystems since the shift is performed for the entire cluster, i.e., all processes within the cluster. This approach is illustrated in Figure 4.3(b). Thus, the requirement in Equation (2.2) is extended as follows:

$$\eta(p, \mathbf{t} - \mathbf{o}_d) + \eta(\tilde{p}, \mathbf{t} - \mathbf{o}_{\tilde{d}}) \leq 1 \tag{4.1}$$

Hence, if a process is shifted by a positive cluster offset, it occupies the resource at a later point in time. In this case, suitable offsets $\mathbf{o}_d$ and $\mathbf{o}_{\tilde{d}}$ have to be found. If a cluster d is shifted by $\mathbf{o}_d$, the previously defined start-times for all messages and tasks are shifted by $\mathbf{o}_d$.

The second technique to obtain a feasible schedule for the entire system is by shifting the start-times of single processes within certain bounds. This approach is illustrated in Figure 4.3(c). The process offset variable $\mathbf{o}_p$ then allows to adjust the process start-time within the cluster. The boundaries for $\mathbf{o}_p$ of a process p with both a predecessor and a successor process are defined as follows.

$$f_{\check{p}} \leq \mathbf{s}_p + \mathbf{o}_p \leq s_{\hat{p}} \tag{4.2}$$

Here, $f_{\check{p}}$ is the latest finish-time of any preceding process $\check{p} \in P_d$ and $s_{\hat{p}}$ is the earliest starting time of any succeeding process $\hat{p} \in P_d$. Thus, the requirement from Equation (4.1) is extended as follows:

$$\eta(p, \mathbf{t} - \mathbf{o}_d - \mathbf{o}_p) + \eta(\tilde{p}, \mathbf{t} - \mathbf{o}_{\tilde{d}} - \mathbf{o}_{\tilde{p}}) \leq 1 \tag{4.3}$$

In this case, additionally, suitable offsets $\mathbf{o}_p$ and $\mathbf{o}_{\tilde{p}}$ have to be found. On the one hand, both approaches help to find a feasible schedule for the entire system. On the other hand, the search space might become very large, in particular with the process shifting approach. In the following, an efficient implementation of this framework is shown.

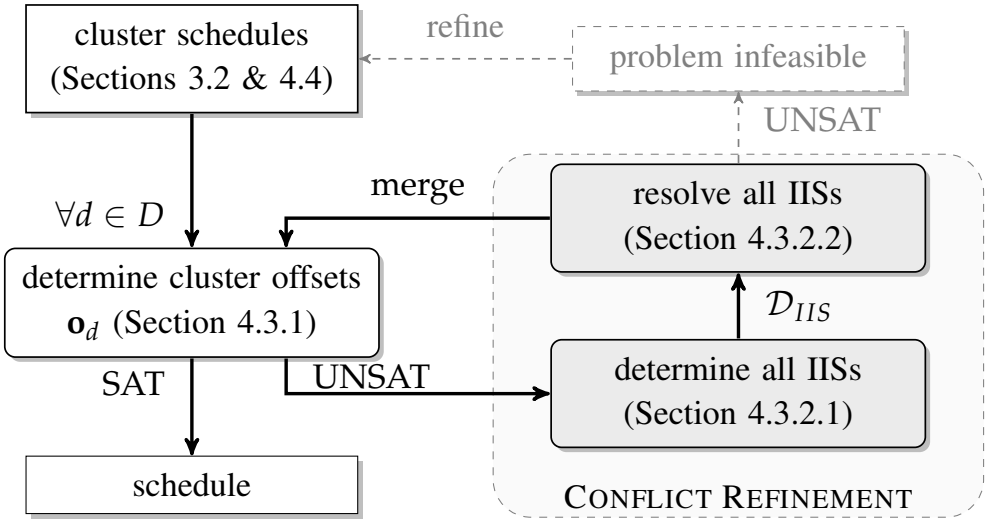


Figure 4.4: *The proposed schedule integration framework first tries to determine a global schedule by only adapting the cluster offsets. If no solution exists, a conflict refinement is applied to resolve infeasibilities in conflicting clusters optimizing both the cluster as well as the process offsets. In case the conflict refinement fails, it provides information to refine the system configuration.*

4.2.2 Schedule Integration Framework

The proposed schedule integration framework merges a set of cluster schedules into a global schedule. For cluster schedule generation we apply the ILP approach presented in Section 3.2. To optimize the integrability of a cluster schedule, Section 4.4 presents various metrics.

For an efficient implementation, the schedule integration is based on multiple stages as illustrated in Figure 4.4. First, the schedule integration approach tries to adapt the start-times s_p of all processes for a cluster d by only using the cluster offset $\mathbf{o}_d$. However, as the general structure for each cluster is fixed, no feasible schedule might exist such that all constraints, defining the offset boundaries, are satisfied (SAT). In case no feasible solution exists for the initial set of clusters (UNSAT), a conflict refinement is required. It adapts the individual process start-times with $\mathbf{o}_p$, while ensuring that the maximum end-to-end timing delay defined for each application is not violated. Therefore, while the conflict refinement modifies the timing behavior of cluster schedules, all previously defined constrains are still valid. Instead of applying the conflict refinement to all clusters, only conflicting cluster schedules are adapted. Therefore, during conflict refinement at first all conflicting cluster schedules in the form of disjunct Irreducible Inconsistent Sets (IISs) are determined. An IIS represents a subset of clusters causing the schedule integration to be infeasible. Once all IISs have been identified, existing conflicts are resolved through adapting individual process start-times. Finally, the adapted clusters are merged into a single cluster (merge) to guarantee termination. The implementation details are described in the sections indicated in Figure 4.4.

This schedule-integration framework allows to design cluster schedules individually which might then be integrated in a global schedule using the proposed framework.

4.3 Integration

This section first presents a schedule integration approach to define an offset for each cluster. Second, a conflict refinement allowing to adapt incompatible cluster schedules is proposed. Finally, extensions to the framework supporting Automotive Ethernet and FlexRay are discussed.

4.3.1 Schedule Integration

During schedule integration, independently created cluster schedules are combined to a global schedule. An offset $\mathbf{o}_d \in \mathbb{R}$ is defined for each cluster schedule $d \in D$ which allows to shift the whole cluster schedule by a constant time. As the cluster schedule is executed with the period $h_d = \operatorname{lcm}_{p \in P_d}(h_p)$[1], the boundaries for $\mathbf{o}_d$ might have any value between 0 and h_d. To significantly reduce the search space, i.e., clearly reducing the runtime for determining offsets for each cluster, a two stage approach is applied: (1) Feasible intervals for the relative offset for each cluster pair $d, \tilde{d} \in D$ are computed. The relative offset is defined as $\mathbf{o}_d - \mathbf{o}_{\tilde{d}}$, hence it defines the relation between the two offsets. (2) The final offsets for the whole set are determined with an SMT approach. Based on Equation (4.1), the set defining all feasible offsets for a single resource r is computed as follows.

$$\begin{aligned} O_{(r,d,\tilde{d})} = \{x | x = \mathbf{o}_d - \mathbf{o}_{\tilde{d}}, \mathbf{o}_d \in [0, h_d], \mathbf{o}_{\tilde{d}} \in [0, h_{\tilde{d}}], p \in P_d, \tilde{p} \in P_{\tilde{d}}, \\ r(p) = r(\tilde{p}) = r, \exists \mathbf{t} : \eta(p, \mathbf{t} - \mathbf{o}_d) + \eta(\tilde{p}, \mathbf{t} - \mathbf{o}_{\tilde{d}}) \leq 1\} \end{aligned} \tag{4.4}$$

Hence, for each process pair p and $\tilde{p}$ that are part of different clusters but are mapped to the same resource, the feasible relative offsets are determined. $O_{(r,d,\tilde{d})}$ represents the intersection of these offsets for all process pairs mapped to resource r, containing all feasible values. Since a cluster is commonly distributed across multiple resources, we need to consider all shared resources in order to compute the relative cluster offset. With $R(d)$ denoting the set of resources used by cluster d, we determine $O_{(d,\tilde{d})}$, defining all feasible values for the relative cluster offset for d and $\tilde{d}$:

$$O_{(d,\tilde{d})} = \bigcap_{r \in R(d) \cap R(\tilde{d})} O_{(r,d,\tilde{d})} \tag{4.5}$$

Now that we have determined all feasible values for the relative offset, the following equation organizes the set of feasible values into intervals $\delta = [\underline{\delta}, \overline{\delta}]$.

[1] Defines the least common multiple of all process periods in cluster d.

$$\Delta_{(d,\tilde{d})} = \{[\underline{\delta}_0, \overline{\delta}_0], [\underline{\delta}_1, \overline{\delta}_1], ..., [\underline{\delta}_k, \overline{\delta}_k]\} \textbf{ with } (\bigcup_{x \in \Delta_{(d,\tilde{d})}} x) = O_{(d,\tilde{d})} \\ \textbf{and } \forall i, j, i \neq j : [\underline{\delta}_i, \overline{\delta}_i] \cap [\underline{\delta}_j, \overline{\delta}_j] = \emptyset \tag{4.6}$$

Based on these intervals, the following SMT formulation determines the cluster offsets: $\forall d \in D$:

$$0 \leq \mathbf{o}_d < h_d \tag{4.7}$$

$\forall d, \tilde{d} \in D, d \neq \tilde{d}$:

$$\bigoplus_{\forall \delta \in \Delta_{(d,\tilde{d})}} \underline{\delta} \leq \mathbf{o}_d - \mathbf{o}_{\tilde{d}} \leq \overline{\delta} \tag{4.8}$$

Constraint (4.7) defines the boundaries for $\mathbf{o}_d$, while Constraint (4.8) ensures that all $\mathbf{o}_d$ lie in the previously determined intervals. Here, the symbol $\oplus$ represents an *exclusive or* (XOR) operation. Updating the process start-times for each cluster with the obtained offsets leads to the global system schedule.

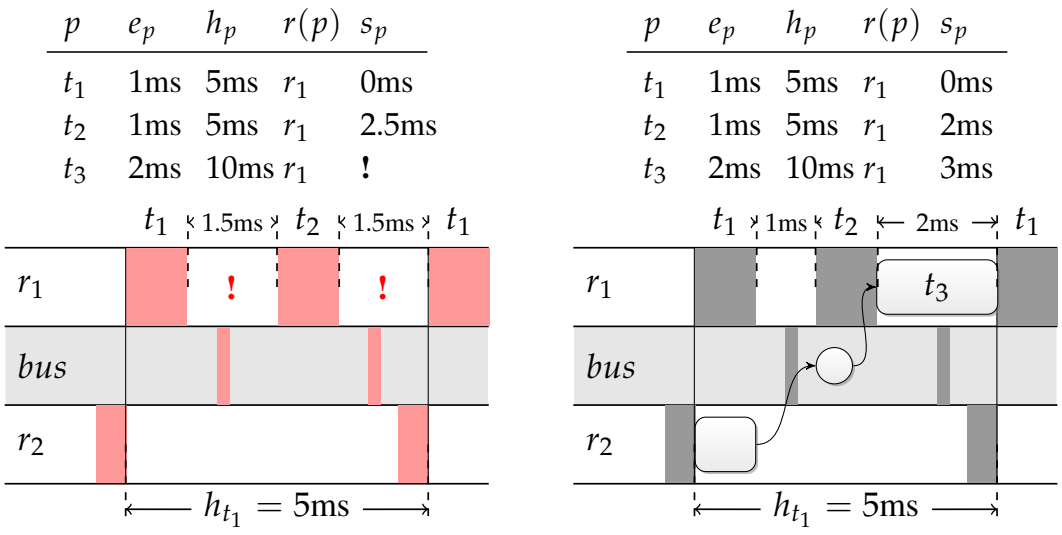
4.3.2 Conflict Refinement

The schedule integration approach assigns an offset to each cluster schedule without altering its structure. However, as each cluster schedule is determined individually, it might not be feasible to integrate all cluster schedules into a global schedule. Figure 4.5 gives an example of two conflicting clusters and how the conflict might be resolved. To overcome such infeasibilities, in the following, a conflict refinement is presented. First, an approach is presented to determine all IISs, representing minimal subsets of clusters which cannot be integrated into a schedule. Second, an approach to resolve conflicts through adapting the initial cluster schedules is proposed.

4.3.2.1 Determine all IISs

While modern SMT solvers might provide the functionality to determine an IIS, these approaches do not allow the use of domain knowledge and analyze each constraint independently. Therefore, we propose an approach tailored to the schedule integration problem. Instead of removing each constraint independently to isolate the constraints leading to the infeasibility, we apply a group-based approach where a whole cluster d, including all affected constraints, is removed from D. The approach is based on the following additional constants and functions:

- Π_d - set of constraints related to cluster d during schedule integration.
- $\mathcal{D}_{IIS}$ - set of IISs, each IIS contains a set of clusters $|D_{IIS}| \geq 2$.



(a) Conflicting cluster schedule

(b) Resolved conflict

Figure 4.5: *Exemplary illustration of conflicting cluster schedules and conflict refinement. (a) The schedule for cluster d_1, containing t_1 and t_2 and illustrated by the grayed out areas, conflicts with the execution-time of t_3, part of cluster d_2. (b) During conflict refinement the start-time of t_2 is adapted to support both clusters in one global schedule.*

- $\beta(p) : P \rightarrow \mathbb{R}$ - schedule defining the start-times of processes. It returns the process start-time s_p for a process $p \in P$.

To determine all IISs, first the domain knowledge obtained during the interval calculation is exploited. If the calculated interval $\Delta_{(d,\tilde{d})}$ is empty, no feasible solution exists. Hence, in a first step, $\mathcal{D}_{IIS}$ is initialized with all pairs of clusters with an empty interval which indicates that both cluster schedules cannot be combined:

$$\mathcal{D}_{IIS} = \{\{d, \tilde{d}\} | d, \tilde{d} \in D, \Delta_{(d,\tilde{d})} = \emptyset\} \tag{4.9}$$

After all initial pair-wise infeasibilities have been determined, an extended deletion filter is applied. While a common deletion filter only determines a single infeasibility [Chi07], here an iterative approach is applied until the remaining set of clusters is feasible. A set of cluster schedules leading to an infeasibility is denoted as $\beta \nvDash \Pi_{\tilde{D}}$[2], indicating that no schedule β could be determined which satisfies all constraints $\Pi_{\tilde{D}}$. In addition, to indicate that all sets in a set of sets $\mathcal{X}$ are merged to a single set we use the notation $X = \bigcup \mathcal{X}$.

Algorithm 4.1 iteratively applies a deletion filter to the problem until the remaining subset $\tilde{D} \subset D$ is feasible (lines 2-10). The deletion filter removes the groups of constraints that affect one cluster iteratively (lines 3-7). If the reduced set remains infeasible (line 4), the cluster is

[2] $x \vDash \Pi$ is read as x *satisfies* Π and $x \nvDash \Pi$ as x *does not satisfy* Π.

Algorithm 4.1: Iterative Deletion Filter extending $\mathcal{D}_{IIS}$ until the remaining subset $\tilde{D}$ is feasible.

Input: set of cluster specifications D for which the schedule integration is unfeasible
Output: set of all subsets of clusters causing an infeasibility $\mathcal{D}_{IIS}$

```
// initialize subset D̃ which should contain the maximal feasible subset
   of clusters
```
1 $\tilde{D} = D \setminus \bigcup \mathcal{D}_{IIS}$
```
// as long as the schedule integration for the clusters in D̃ is
   intractable continue
```
2 **while** $\beta \nvDash \Pi_{\tilde{D}}$ **do**
3 | **for** $d \in \tilde{D}$ **do**
| | `// if the problem is still infeasible after the constraints for d have been removed, remove d from D̃`
4 | | **if** $\beta \nvDash \Pi_{\tilde{D}} \setminus \Pi_d$ **then**
5 | | | $\tilde{D} = \tilde{D} \setminus d$
6 | | **end**
7 | **end**
8 | $\mathcal{D}_{IIS} = \{\tilde{D}\} \cup \mathcal{D}_{IIS}$
9 | $\tilde{D} = D \setminus \bigcup \mathcal{D}_{IIS}$
10 **end**

removed from the infeasible set (line 5), otherwise it remains part of it. After an IIS has been determined, it is added to $\mathcal{D}_{IIS}$ (line 8), and the process is repeated for an updated $\tilde{D}$ (line 9).

4.3.2.2 Resolve all IISs

Once all conflicting cluster schedules have been determined, a conflict refinement is required to resolve all conflicts. This section proposes an SMT-based approach to adapt the start-times of individual processes in addition to the cluster offset. In the following, first a problem partitioning is proposed to determine which IISs need to be resolved concurrently. Second, we discuss how the process offset intervals might be selected. Finally, the conflict refinement approach is presented.

Problem partitioning. To resolve the conflicts determined in the set of IISs, various constraints have to be taken into account, i.e., an offset has to be determined for each process and for each cluster concurrently. In contrast to determining cluster offsets (Section 4.3.1), the conflict refinement might therefore suffer from a limited scalability. However, if multiple IISs have been determined which do not concern the same clusters, the IISs can be addressed individually. We therefore propose a partitioning heuristic that identifies suitable partitions for the set of IISs $\mathcal{D}_{IIS}$ to improve the scalability.

Automotive E/E-architectures are partitioned into domains, thus, different clusters might only share few or no resources. Hence, if IISs are distributed over different domains, resolving each individually is feasible, while IISs concerning clusters from the same domain might require an additional conflict refinement if not resolved concurrently. Figure 4.6(a) shows a system which implements four clusters in two domains. The clusters in the two domains only share resource r_4. Hence, if in each domain an IIS was determined, resolving each IIS individually and integrating the resolved schedules into the global schedule is likely to lead to a feasible solution. The partitioning is defined as:

$$\{\tilde{\mathcal{D}}_{IIS}, ...\} = \text{partition}(\mathcal{D}_{IIS})$$

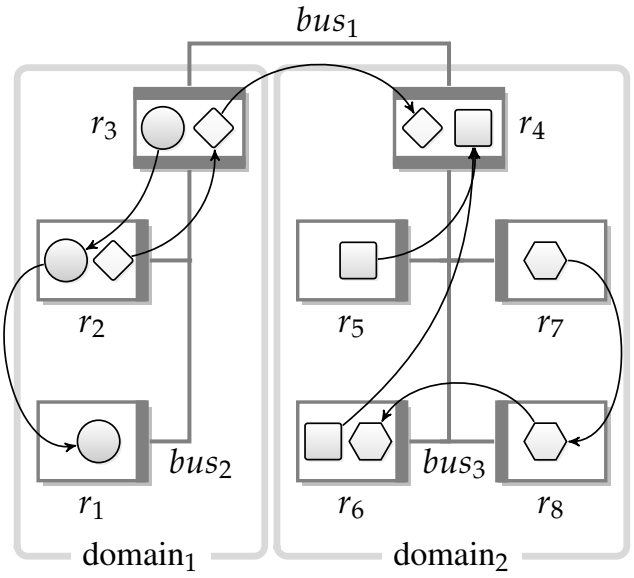
$$\tilde{\mathcal{D}}_{IIS} \subseteq \mathcal{D}_{IIS}$$

IISs are strongly correlated if they share multiple resources and the feasible cluster offsets are strongly limited. The feasible cluster offset relation, as defined in Equation (4.6), quantifies this relation well. For instance, if two clusters do not share any resource, the feasible cluster has the maximal interval length $\Delta_{(d,\tilde{d})} = \{[-h_{\tilde{d}}, h_d]\}$. Based on the feasible cluster offsets $\Delta_{(d,\tilde{d})}$, we therefore introduce the following metric to determine the correlation of two IISs D and $\tilde{D}$:

$$\mu_{\text{correlation}}(D, \tilde{D}) = \underbrace{\frac{1}{\sqrt{|D| + |\tilde{D}|} - 1}}_{\substack{\text{scale to number} \\ \text{of clusters}}} \cdot \sum_{d \in D} \sum_{\tilde{d} \in \tilde{D}} \left(1 - \underbrace{\frac{\sum_{\delta \in \Delta_{(d,\tilde{d})}} (\overline{\delta} - \underline{\delta})}{h_d + h_{\tilde{d}}}}_{\substack{\text{percentage of} \\ \text{feasible offset} \\ \text{ratios}}} \right) \tag{4.10}$$

The metric determines the percentage of feasible offset ratios by calculating the sum of all intervals divided by the maximal interval length, i.e., if two clusters do not share any resources. We subtract this value from 1 to obtain a value of 0 if two clusters do not share any resources, and a value of 1 if both clusters conflict, i.e., no feasible cluster offset combination exists. The metric forms the sum of all these cluster relations between two IISs, determining the correlation between both. As IISs might strongly differ in the number of clusters, we finally relate this sum to the number of clusters in the IISs. We apply the square root to the number of clusters and subtract 1 to normalize the metric value for the smallest IIS size of 2. Figure 4.6(b) gives an example on how $\mu_{\text{correlation}}(D, \tilde{D})$ is determined for the system illustrated in Figure 4.6(a). Based on this correlation metric, the partitioning heuristic partitions the set of all IISs in subsets with a low correlation between each other.

As a cluster might be part of multiple IISs, the partitioning heuristic starts with identifying IISs which contain common clusters. These IISs have to be handled concurrently, independent of the correlation value determined by the metric. Once these extended IISs have been identified, we apply our correlation metric to all IIS pairs. Two IISs are part of the same partition if their correlation is above the threshold $\epsilon_{\text{correlation}}$, i.e., $\mu_{\text{correlation}}(D, \tilde{D}) > \epsilon_{\text{correlation}}$. If all



(a) Domain-based architecture

CLUSTERS

d_1: ○ d_2: ◇
d_3: □ d_4: ⬡

CONFLICTS

$D_{IIS} = \{d_1, d_2\}$
$\tilde{D}_{IIS} = \{d_3, d_4\}$

$\Delta_{(d_1,d_3)} = \{[-5,5]\}$
$\Delta_{(d_1,d_4)} = \{[-5,5]\}$
$\Delta_{(d_2,d_3)} = \{[-5,-1],[0,4]\}$
$\Delta_{(d_2,d_4)} = \{[-5,5]\}$

$$\mu_{\text{correlation}}(D_{IIS}, \tilde{D}_{IIS}) = \\ = \frac{1}{\sqrt{2+2}-1} \cdot (1 - \frac{10}{10} + 1 - \frac{10}{10} + \\ + 1 - \frac{4+4}{10} + 1 - \frac{10}{10}) = \\ = 1 \cdot (0 + 0 + 0.2 + 0) = 0.2$$

(b) Partitioning metric

Figure 4.6: *(a) Automotive systems are realized by a domain-based architecture where clusters distributed over different domains only share few resources. (b) The partitioning metric determines the correlation between two cluster sets D_{IIS} and $\tilde{D}_{IIS}$ based on the feasible relative cluster offsets $\Delta_{(d,\tilde{d})}$. For this example the cluster period for all clusters is* 5 ms.

IISs share a strong correlation, the outcome of this metric might be that the conflict refinement should consider all IISs concurrently. The conflict refinement is then applied for each partition individually, before the partition schedules are integrated in a global schedule with the approach presented in Section 4.3.1.

Determine intervals for process offsets. The conflict refinement adapts the process offsets for conflicting cluster schedules. To ensure all constraints defined for the cluster schedule are not affected by adapting process start-times, the conflict refinement must ensure that the previously defined constraints such as the maximum end-to-end delay are not violated. Thus, suitable process offsets must be determined. For intermediate processes, i.e., processes having both a predecessor and a successor, this means that the selected process offset ensures that the precedence constraints are satisfied. As stated in Equation 4.2, the process offset $\mathbf{o}_p$ for a process p is limited by the finish-time of a predecessor task $\check{p}$ it receives data from, and the start-time of a successor task $\hat{p}$ processing data sent by p. Figure 4.7 illustrates the offsets for the intermediate process m_5. Its predecessor $\check{m}_5 = t_1$ and its successor $\hat{m}_5 = t_2$ define the boundaries for $\mathbf{o}_{m_5}$ to $\delta_{o_{m_5}} = [-w_{t_1,m_5} + \mathbf{o}_{t_1}, w_{m_5,t_2} + \mathbf{o}_{t_2}]$. Here, $w_{(p,\tilde{p})}$ denotes the waiting-time between data-dependent processes. Adapting the process offset of intermediate processes such that the

precedence constraints are satisfied, does not affect the end-to-end delay of an application and allows to determine a valid schedule.

To determine suitable intervals for each process offset, the following additional parameters are introduced:

- $\delta_{\mathbf{o}_p} = [\underline{\delta_{\mathbf{o}_p}}, \overline{\delta_{\mathbf{o}_p}}]$ - interval defining the lower and upper bound for the process offset.
- $\underline{\delta_{o_{p_{src}}}}$, $\overline{\delta_{o_{p_{snk}}}}$ - lower bound of a source and upper bound of sink process, respectively. A source process does not have a predecessor while a sink process does not have a successor. See Definitions 2.2.2 and 2.2.3.
- $w_{(p,\hat{p})} = s_{\hat{p}} - f_p$ - waiting-time from the finish-time of p to the start-time of the data-dependent successor process $\hat{p}$.
- $a \in A_d$ - defines an application with its task graph $G_a = (P_a, E_a)$. Described in detail in Definition 2.1.2.
- $\check{E}(\phi, p) : \Phi, P \rightarrow \mathcal{E}$ - returns all edges in the path ϕ from the source process to p.
- $\hat{E}(\phi, p) : \Phi, P \rightarrow \mathcal{E}$ - returns all edges in the path ϕ from p to the sink process.
- $\mathrm{src}(\phi), \mathrm{snk}(\phi) : \Phi \rightarrow P$ - the functions return the source and the sink process in path ϕ, respectively.

The offset interval for a process $\mathbf{o}_p \in \delta_{\mathbf{o}_p} = [\underline{\delta_{\mathbf{o}_p}}, \overline{\delta_{\mathbf{o}_p}}]$ can then be defined by the following boundaries.
$\forall d \in \mathcal{D}_{IIS}, p \in P_d, \phi \in \{x | x \in \Phi(E_a), \exists x : p \in x\}$,
$p_{src} = \mathrm{src}(\phi), p_{snk} = \mathrm{snk}(\phi)$:

$$\underline{\delta_{\mathbf{o}_p}} = \operatorname*{argmax}_{\phi} \Big(\underline{\delta_{o_{p_{src}}}} - \sum_{(\tilde{p},\hat{\tilde{p}}) \in \check{E}(\phi,p)} w_{(\tilde{p},\hat{\tilde{p}})} \Big) \tag{4.11}$$

$$\overline{\delta_{\mathbf{o}_p}} = \operatorname*{argmin}_{\phi} \Big(\overline{\delta_{o_{p_{snk}}}} + \sum_{(\tilde{p},\hat{\tilde{p}}) \in \hat{E}(\phi,p)} w_{(\tilde{p},\hat{\tilde{p}})} \Big) \tag{4.12}$$

Constraint (4.11) defines the lower bound for the process offset. It is defined by the sum of all waiting-times between data-dependent processes in a path and the offset defined for the source process. As a process might have multiple source or sink processes, several paths might have to be taken into account and the strictest lower bound, i.e., the largest value, is selected. Constraint (4.12) defines the upper bound for $\mathbf{o}_p$ accordingly. For the upper bound the strictest bound is defined by the lowest value for all paths. The interval $\delta_{\mathbf{o}_p} = [\underline{\delta_{\mathbf{o}_p}}, \overline{\delta_{\mathbf{o}_p}}]$ then contains all feasible offset values that still satisfy all precedence and end-to-end-delay constraints.

In addition to adjusting the offsets of intermediate processes, in some cases it might also be necessary to adjust the offsets of source or sink processes, e.g., if the initial cluster schedules define a tight end-to-end delay. For the source and sink process offsets, we suggest two options to

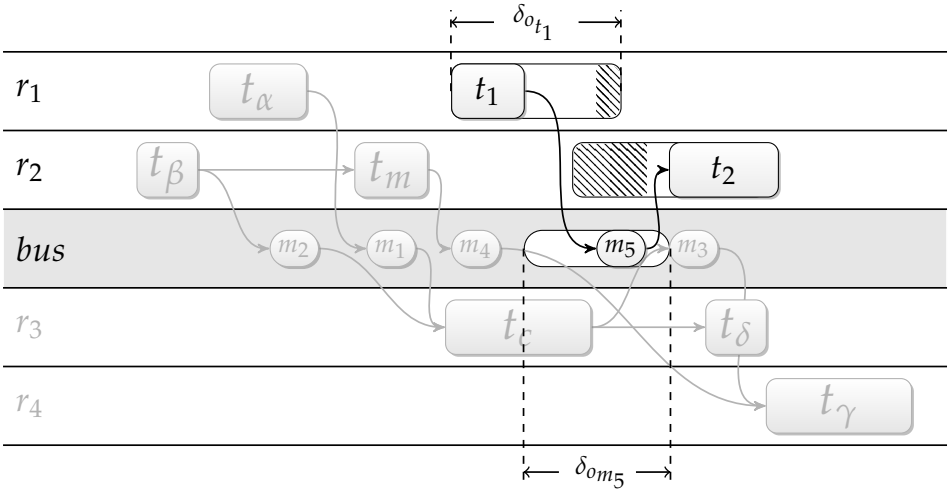


Figure 4.7: *During conflict refinement the process offsets are adapted individually. The illustration shows the intervals for the process offsets if the end-to-end delay of the cluster schedule is fixed. This is only illustrated for one of the two applications. Here, the white areas indicate the waiting-time between data-dependent processes. As the offset optimization of all processes is done concurrently, the offset interval also depends on the offset of predecessor and successor tasks (▨).*

define the boundaries of $\mathbf{o}_p$. (1) Keep all end-to-end delays as defined for subsystem schedules. (2) Allow adapting the end-to-end delay up to the maximum end-to-end delay θ_a. Both options ensure that the timing requirements defined by the application designer are satisfied. Depending on the subsystem requirements, for each cluster or application the more suitable option should be selected. For instance, if an application does not benefit from a minimal end-to-end delay, option (2) allows a more flexible conflict refinement.

For option (1), the end-to-end delay of an application might be tightened, i.e., the end-to-end delay can be decreased, but not increased. Therefore, the source and sink processes might only be shifted in one direction. The lower bound for the source process $\underline{\delta_{o_{p_{src}}}}$, and the upper bound for the sink process $\overline{\delta_{o_{p_{snk}}}}$ are then defined as:

$$\underline{\delta_{o_{p_{src}}}} = 0 \tag{4.13}$$

$$\overline{\delta_{o_{p_{snk}}}} = 0 \tag{4.14}$$

For instance, see processes t_1 and t_2 in Figure 4.7. The boundaries for the source process t_1 are defined as $\mathbf{o}_{t_1} \in [0, w_{t_1,m_5} + \mathbf{o}_{m_5}]$. Similarly, for the sink process t_2: $\mathbf{o}_{t_2} \in [-w_{m_5,p_2} + \mathbf{o}_{m_5}, 0]$.

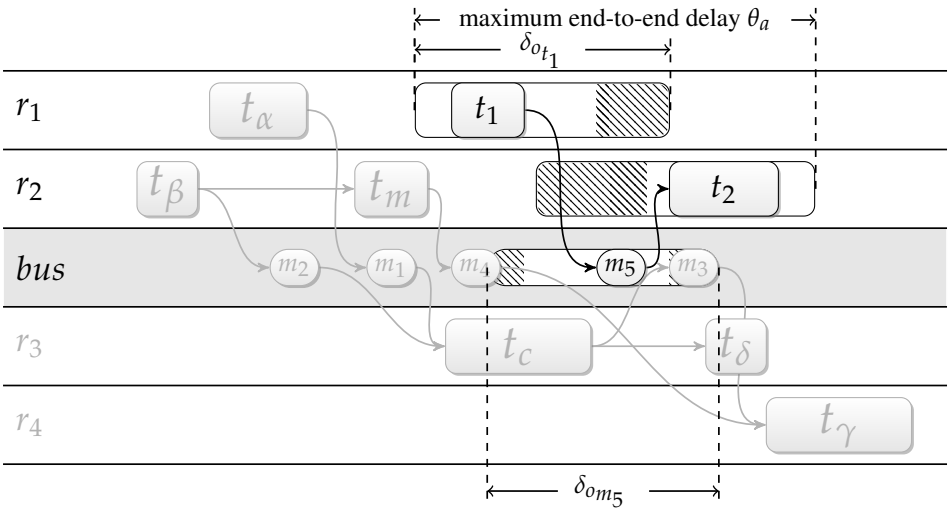


Figure 4.8: *Illustration of intervals for the process offsets if the offsets for source and sink processes might increase the end-to-end delay up to the maximum end-to-end delay defined for an application. Here, the white areas indicate the offset if predecessor or successor tasks are not shifted, while the areas indicated by ▨ indicate the dependency on the offsets assigned to these processes.*

For option (2), the maximum end-to-end delay of an application might be increased up to the full maximum end-to-end delay. To be able to evenly distribute the additional slack or laxity, we introduce the following function.

- $\mathrm{ctr}(a) : A \rightarrow \mathbb{R}$ - returns the temporal center of an application, i.e., the time instant forming the median between the process with the earliest start-time and the process with the latest finish-time.

The lower bound for the source process $\underline{\delta_{o_{p_{src}}}}$, and the upper bound for the sink process $\overline{\delta_{o_{p_{snk}}}}$ can then be defined as:

$$\underline{\delta_{o_{p_{src}}}} = s_{(p=\mathrm{src}(\phi))} - \theta_a + \mathrm{ctr}(a) \tag{4.15}$$

$$\overline{\delta_{o_{p_{snk}}}} = s_{(p=\mathrm{snk}(\phi))} + \theta_a - \mathrm{ctr}(a) \tag{4.16}$$

Constraint (4.15) determines the temporal center between the start of all source processes and the termination of all sink processes with the function $\mathrm{ctr}(a)$. Based on this central time instant, the feasible lower bound for the source process is determined. This offset is evenly distributed between source and sink processes. Similarly, Constraint (4.16) determines the upper bound for sink processes. Figure 4.8 illustrates the intervals for the process offsets of an application, if the end-to-end delay is allowed to be increased during conflict refinement.

SMT-based approach. Once suitable intervals have been determined for each offset, we apply an SMT-based conflict refinement. The conflict refinement is applied to each of the previously defined partitions $\tilde{\mathcal{D}}_{IIS} \subseteq \mathcal{D}_{IIS}$ individually. We therefore merge the set of IISs $\tilde{\mathcal{D}}_{IIS}$ to a single set of clusters $D_{IIS} = \bigcup \tilde{\mathcal{D}}_{IIS}$. The conflict refinement is then formulated as follows.
$\forall d \in D_{IIS}$:

$$0 \leq \mathbf{o}_d < h_d \tag{4.17}$$

$\forall d \in D_{IIS}, \forall p \in P_d, \delta_{\mathbf{o}_p} = [\underline{\delta_{\mathbf{o}_p}}, \overline{\delta_{\mathbf{o}_p}}]$:

$$\underline{\delta_{\mathbf{o}_p}} \leq \mathbf{o}_p \leq \overline{\delta_{\mathbf{o}_p}} \tag{4.18}$$

$\forall d, \tilde{d} \in D_{IIS}, \forall p \in P_d, \forall \tilde{p} \in P_{\tilde{d}}, p \neq \tilde{p}, r(p) = r(\tilde{p}),$
$i = \{0, .., \frac{3 \cdot h(p,\tilde{p})}{h_p} - 1\}, j = \{0, .., \frac{3 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\}$:

$$\begin{aligned} &\mathbf{o}_d + \mathbf{o}_p + i \cdot h_p + s_p + \tau_p \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o}_{\tilde{p}} + \mathbf{o}_{\tilde{d}} \\ \oplus \quad &\mathbf{o}_d + \mathbf{o}_{\tilde{p}} + j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \tau_{\tilde{p}} \leq i \cdot h_p + s_p + \mathbf{o}_p + \mathbf{o}_d \end{aligned} \tag{4.19}$$

$\forall d \in D_{IIS}, a \in A_d, \phi \in \Phi(E_a), (p, \hat{p}) \in \phi$:

$$\mathbf{o}_p \leq \mathbf{o}_{\tilde{p}} + w_{(p,\hat{p})} \tag{4.20}$$

Constraint (4.17) and Constraint (4.18) define the boundaries for the cluster offsets and the process offsets, respectively. Constraint (4.19) ensures that two processes which are mapped to the same resource do not intersect for the selected cluster and process offsets, i.e., that one of the processes finishes its execution before the other process is started (cf. Requirement 2.2.1). Note that this constraint also includes processes that are part of the same cluster. Finally, Constraint (4.20) ensures that the precedence constraints (cf. Requirement 2.2.2) are not violated. Feasible values for the process offset must therefore ensure that the assigned offset only shifts the process p up to the adapted start-time of the successor task $\hat{p}$.

After the conflict refinement has determined a valid schedule, all clusters in D_{IIS} are merged to a single cluster d_{IIS} to prevent repeated optimization and ensure termination of the algorithm.

Reduce the need for a conflict refinement. Compared to the schedule integration solely defining an offset for each cluster, the conflict refinement is computationally more expensive, as various additional constraints need to be taken into account. Thus, ideally the number of conflicts should be low. Conflicts are in particular likely if each cluster consists of multiple applications. However, as applications are generally independent of each other, it is often not necessary to assign the same cluster offset to all applications. If this is the case, considering each application

as an individual cluster clearly decreases the probability that a conflict refinement needs to be applied.

4.3.3 Automotive Ethernet

The schedule integration and the conflict refinement presented in the previous sections might be directly applied for Automotive Ethernet relying on TSN for a time-triggered communication. However, as these formulations do not consider the switched network and a full-duplex communication, concurrent message transmission would be omitted and the bandwidth requirements overestimated. To support these Ethernet properties we therefore introduce the following parameters.

- $r_{eth} \in R_{eth}$ - resource representing an Ethernet bus. It holds that all resources $r \in R_{eth}$ represent an Ethernet bus.
- $\psi(p)$ - function determining all routings for a message sent on the Ethernet bus, i.e., it returns all paths containing the data links that connect the sending resource to all receiving resources. For instance, $\psi(p) = \{(r, sw_1), (sw_1, sw_2), (sw_2, \tilde{r})\}$ describes the routing from resource r to $\tilde{r}$ via the switches sw_1 and sw_2.

Based on $\psi(p)$, the Equation (4.4) defining the relative cluster offsets for each resource $O_{(r,d,\tilde{d})}$ can be adapted to support Ethernet:

$$\begin{aligned} O_{(r_{eth},d,\tilde{d})} = \{x | x = \mathbf{o}_d - \mathbf{o}_{\tilde{d}}, \mathbf{o}_d \in [0, h_d], \mathbf{o}_{\tilde{d}} \in [0, h_{\tilde{d}}], p \in P_d, \tilde{p} \in P_{\tilde{d}}, \\ r(p) = r(\tilde{p}) = r_{eth}, \psi(p) \cap \psi(\tilde{p}) \neq \emptyset, \exists \mathbf{t} : \eta(p, \mathbf{t} - \mathbf{o}_d) + \eta(\tilde{p}, \mathbf{t} - \mathbf{o}_{\tilde{d}}) \leq 1\} \end{aligned} \tag{4.21}$$

The relative offset is only considered for messages which share at least one link between switches and/ or resources in the same direction. Similarly, for Constraint (4.19) in the conflict refinement, the conditions can be redefined for Ethernet messages:

$$\begin{aligned} \forall d, \tilde{d} \in D_{IIS}, \forall p \in P_d, \forall \tilde{p} \in P_{\tilde{d}}, p \neq \tilde{p}, r(p) = r(\tilde{p}) \in R_{eth}, \\ \psi(p) \cap \psi(\tilde{p}) \neq \emptyset, i = \{0, .., \tfrac{3 \cdot h(p,\tilde{p})}{h_p} - 1\}, j = \{0, .., \tfrac{3 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\} : \end{aligned} \tag{4.22}$$

If these constraints are applied for all Ethernet resources, support for full-duplex switched Ethernet networks is added to our schedule integration framework.

4.3.4 FlexRay

This section presents an extension of our framework to support schedule integration for the FlexRay bus. Section 3.1.3.2 proposed a schedule integration approach for FlexRay message scheduling in an asynchronous system. While this schedule integration approach might be integrated into our framework, it would require the introduction of various additional constraints

to integrate the binary variables. Instead, we have selected a more efficient approach which only adds few additional constraints to the general schedule integration framework introduced in Sections 4.3.1 and 4.3.2. The additional constraints are based on the following constants.

- sl_m - slot message m is assigned to in the cluster schedule.
- b_m - base-cycle assigned to m in the cluster schedule.
- n_{fr} - number of slots in static FlexRay segment.
- τ_{fr} - duration of a static FlexRay slot.
- h_{fr} - period of a FlexRay cycle.
- $\check{r}(m) : M \rightarrow R$ - ECU sending the message m.
- R_{fr} - set of all resources representing a FlexRay bus.

A FlexRay schedule assigns a slot and base-cycle to each message, while the schedule integration framework assigns start-times to processes which are defined by their execution-time. The relation of the FlexRay message parameters to this temporal representation might be defined as follows (cf. Requirement 2.3.1).
$\forall m \in \{p | p \in P_d, \exists p : r(p) \in R_{\text{fr}}\}$:

$$\tau_m = \tau_{\text{fr}} \tag{4.23}$$

$$s_m = sl_m \cdot \tau_{\text{fr}} + b_m \cdot h_{\text{fr}} \tag{4.24}$$

For the transmission time defined in Constraint (4.23) we assume the duration of a static FlexRay slot. Note that this slot might still contain several messages, but as all data needs to be available at the beginning of the slot and is only available once the whole slot was received, the transmission time perceived by data-dependent processes is therefore the slot duration. Constraint (4.24) relates the message start-time to its slot and base-cycle assignment.

The FlexRay static segment is partitioned into n_{fr} slots to which messages can be assigned. We assume that the first static slot starts at the time instant 0 for all cluster schedules (cf. Definition 2.3.1). The start-times of all FlexRay messages must equal the start-time of one of the slots. As FlexRay also supports additional segments, e.g., the dynamic segment, this requirement also ensures that no messages are scheduled outside the static segment. For the schedule integration approach in Section 4.3.1, for each FlexRay message we can define feasible cluster offsets:
$\forall m \in \{p | p \in P_d, \exists p : r(p) \in R_{\text{fr}}\}$:

$$\delta^{\text{fr}}_{(d,m)} = [0, \tau_{\text{fr}}, ..., (n_{\text{fr}} - 1 - sl_p) \cdot \tau_{\text{fr}}, ..., h_{\text{fr}} - sl_p \cdot \tau_{\text{fr}}, ..., h_{\text{fr}}, h_{\text{fr}} + \tau_{\text{fr}}, ..., h_d - \tau_{\text{fr}}] \tag{4.25}$$

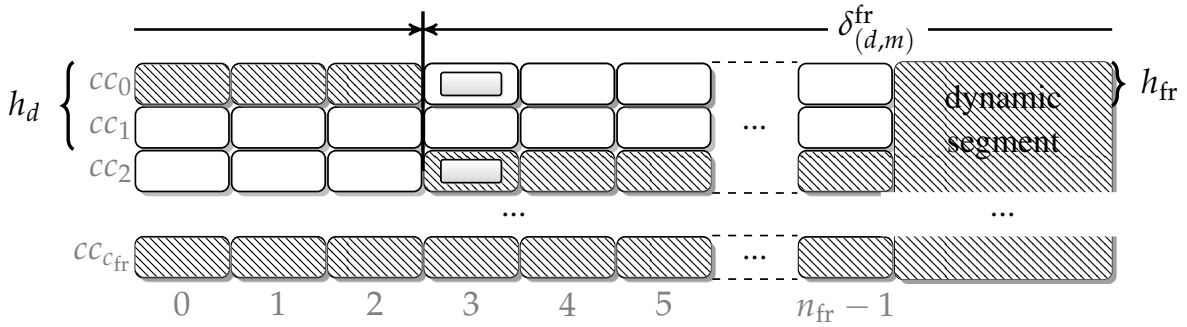


Figure 4.9: *Offset interval determination for a message scheduled on the FlexRay bus. Feasible values for the cluster offset are start-times of all slots illustrated in white.*

The cluster offset can only be adjusted in steps of the slot duration of τ_{fr}. Depending on the slot m is assigned to, feasible offset values are defined to ensure that messages are only shifted within the static segment. Figure 4.9 gives an exemplary illustration of this interval. For the schedule integration, the intervals for a cluster offset $\mathbf{o}_d$ might then be defined as the intersection of all FlexRay messages of a cluster:

$$\delta^{\text{fr}}_d = \bigcap_{\substack{\forall m \in \{p | p \in P_d, \\ \exists p : r(p) \in R_{\text{fr}}\}}} \delta^{\text{fr}}_{(d,m)} \tag{4.26}$$

Based on these cluster offsets, the relative cluster offset determination from Equation (4.4) can be adapted to support FlexRay as follows.
$M_d = \{p | p \in P_d, \exists p : r(p) \in R_{\text{fr}}\}, M_{\tilde{d}} = \{p | p \in P_{\tilde{d}}, \exists p : r(p) \in R_{\text{fr}}\}$:

$$\begin{aligned} O_{(r_{\text{fr}},d,\tilde{d})} = \{x | x = \mathbf{o}_d - \mathbf{o}_{\tilde{d}}, \mathbf{o}_d \in \delta^{\text{fr}}_d, \mathbf{o}_{\tilde{d}} \in \delta^{\text{fr}}_{\tilde{d}}, m \in M_d, \tilde{m} \in M_{\tilde{d}}, \\ r(m) = r(\tilde{m}) = r_{\text{fr}}, \exists \mathbf{t} : \eta(m, \mathbf{t} - \mathbf{o}_d) + \eta(\tilde{m}, \mathbf{t} - \mathbf{o}_{\tilde{d}}) \leq 1\} \end{aligned} \tag{4.27}$$

The function determines feasible relative offsets, taking all FlexRay constraints into account. In addition, also the conflict refinement (see Section 4.3.2) requires additional constraints to support FlexRay. We assume it holds that the duration of a FlexRay cycle h_{fr} is a multiple of the duration of a static slot, thus, $h_{\text{fr}} \bmod \tau_{\text{fr}} = 0$. We can then define the theoretical number of static slots in a FlexRay cycle if the dynamic segment, symbol window and network idle time are included:

$$n_{all} = \frac{h_{\text{fr}}}{\tau_{\text{fr}}} \tag{4.28}$$

In addition, we introduce the following variable:

- $\mathbf{i}_m \in \mathbb{N}_0$ - integer variable indicating to which slot and base-cycle a FlexRay message is mapped to

For the SMT approach in the conflict refinement, we then define the following additional constraints for FlexRay messages, extending the Constraints (4.17)- (4.20) (page 92):
$\forall d \in D, m \in \{p | p \in P_d, \exists p : r(p) \in R_{\text{fr}}\}$:

$$s_m + \mathbf{o}_d + \mathbf{o}_m = \mathbf{i}_m \cdot \tau_{\text{fr}} \tag{4.29}$$

$$\bigoplus_{b \in \{0,..,\frac{h_m}{h_{\text{fr}}}-1\}} b \cdot n_{all} \leq \mathbf{i}_m < b \cdot n_{all} + n_{\text{fr}} \tag{4.30}$$

Constraint (4.29) ensures that the start-time of a FlexRay message equals the start-time of a static slot. Constraint (4.30) defines the boundaries for $\mathbf{i}_m$, ensuring that messages are only mapped to static slots.

For the FlexRay extension to our schedule integration framework, we assume that messages of different clusters do not share the same slot in a cycle and therefore do not consider a repacking of messages to slots.

FlexRay 2.1. The FlexRay extensions discussed in this section are suitable for FlexRay 3.0 scheduling which allows to share a slot between different ECUs. To support FlexRay 2.1, the schedule integration must ensure that a slot is exclusively used by one ECU. For both the schedule integration as well as the conflict refinement, this constraint can be efficiently implemented by introducing a hypothetical process period h_m^{fr} of the duration of a FlexRay cycle:
$\forall d, \tilde{d} \in D, m \in \{p | p \in P_d, \exists p : r(p) \in R_{\text{fr}}\}, \tilde{m} \in \{p | p \in P_{\tilde{d}}, \exists p : r(p) \in R_{\text{fr}}\}$,
$m \neq \tilde{m}, r(m) = r(\tilde{m})$:

$$h_m^{\text{fr}}, h_{\tilde{m}}^{\text{fr}} = \begin{cases} h_{\text{fr}}, h_{\text{fr}} & \check{r}(m) \neq \check{r}(\tilde{m}) \\ h_m, h_{\tilde{m}} & \text{otherwise} \end{cases} \tag{4.31}$$

If the messages are not sent from the same resource, instead of using the actual process period, smaller periods equal to the duration of a FlexRay cycle are selected. This ensures that messages sent by different resources are never mapped to the same slot (cf. Requirement 2.3.2). Hence, replacing the period of a FlexRay message with h_m^{fr}, according to Equation (4.31), allows to apply the schedule integration framework for FlexRay 2.1 scheduling.

4.4 Metrics for Subsystem Scheduling

The previous sections were concerned with the integration of cluster schedules into a single global schedule. The integrability of different cluster schedules into a common schedule and

the need for a conflict refinement depends on the structure of each cluster schedule. In the following, we therefore discuss several metrics that allow to optimize cluster schedules for schedule integration. The focus lies on optimizing the individual cluster schedules without having knowledge about other clusters the schedule should be integrated with. The proposed metrics extend the ILP approach presented in Section 3.2. The metrics can be grouped in two types: (1) Metrics to increase the variability of single processes, i.e., increasing the degree to which single process offsets might be adapted without affecting other processes. (2) Metrics to optimize the cluster schedule such that the idle times between the activation of processes are cumulated. In the following, we first propose two metrics falling in group (1): A metric to maximize the slack usage of applications and a metric to evenly distribute processes on resources. Finally, we propose a metric for optimizing the cluster schedule for the maximal continuous idle time, which falls in group (2).

4.4.1 Maximize Slack Usage

Applications in the automotive domain define maximum end-to-end delays below which a correct functionality is guaranteed. While minimizing the end-to-end delay might improve the performance, it is often not required, as satisfying the maximum end-to-end delay already ensures a correct and accurate behavior. To improve the integrability of a cluster we therefore propose a metric which maximizes the slack usage of applications. The slack defines the time difference between the maximum end-to-end delay and the actual end-to-end delay of an application. The goal of the metric is to introduce an evenly distributed idle time between data-dependent processes. This allows to adjust the start-time of single processes without having to adjust the start-times of the predecessor and successor processes. The metric is based on the following variables:

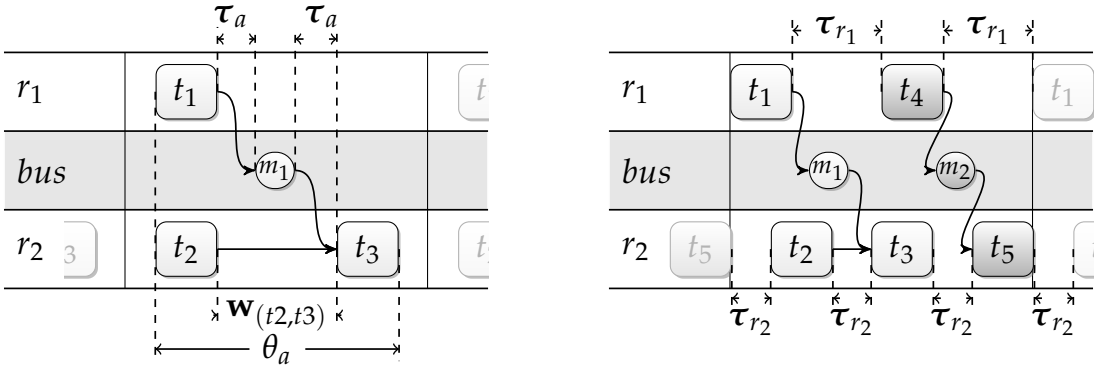
- $\mathbf{w}_{(p,\tilde{p})} \in \mathbb{R}$ - variable for waiting-time between the two processes p and $\tilde{p}$.
- $\boldsymbol{\tau}_a \in \mathbb{R}$ - variable for minimal waiting-time between processes of application a.

Figure 4.10(a) illustrates the metric measuring the slack utilization of an application. The ILP formulation in Section 3.2.2 is then extended with the following constraint.
$a \in A_d, e \in E_a, (p, \tilde{p}) \in e$

$$\boldsymbol{\tau}_a \leq \mathbf{w}_{(p,\tilde{p})} \tag{4.32}$$

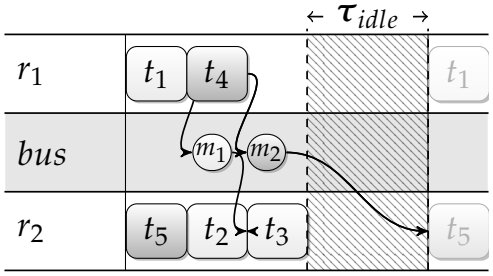
The constraint defines a lower bound for the waiting-time between data-dependent processes. Based on this lower bound, we define the following objective.

$$\max\left(\sum_{e \in E_d, (p,\tilde{p}) \in e} \mathbf{w}_{(p,\tilde{p})} + \sum_{a \in A_d} \boldsymbol{\tau}_a\right) \tag{4.33}$$



(a) Measure slack usage

(b) Evenly distribute processes



(c) Cumulate idle time

Figure 4.10: *Illustration of proposed metrics to evaluate the integrability of a cluster schedule. (a) Evaluates the integrability by the waiting-time between processes and the spread of this waiting-time. The upper bound for the process distribution is the maximum end-to-end delay θ_a. (b) Measures the distribution of processes on each resource. (c) Evaluates the integrability by the longest available idle segment over all resources.*

The objective maximizes the overall waiting-time between all data-dependent processes and, thus, maximizes the slack usage of the cluster schedule. In addition, the objective also considers the lower bound for the waiting-time and therefore ensures that a minimal delay is introduced between all data-dependent processes. Thus, based on this metric, the idle time between all data-dependent processes of an application might be evenly distributed.

4.4.2 Evenly Distribute Processes

The previous subsection proposed a metric to increase the variability of processes within one application. However, the variability of a process does not only depend on its data-dependent predecessor and successor tasks, but also on the processes it shares a resource with. To improve the variability on each resource we therefore propose a metric which evenly distributes

processes on a resource. This does not only improve the variability, but also might facilitate the integration of additional processes as the idle segments are evenly distributed. The following additional constants and variables are introduced:

- $h(p, \tilde{p}) : P \rightarrow \mathbb{R}$ - returns the hyper-period of the two processes p and $\tilde{p}$.
- $\boldsymbol{\tau}_r \in \mathbb{R}$ - variable for lower bound of minimal idle-time between processes on resource r.
- $\bar{\mathbf{x}}_{(p,\tilde{p})} \in \{0, 1\}$ - binary variable indicating if p is started before $\tilde{p}$.

Figure 4.10(b) illustrates the metric measuring the even distribution of processes on each resource. Based on these variables, we can then define the following additional constraints for the ILP formulation to determine cluster schedules.
$p, \tilde{p} \in P_d, p \neq \tilde{p}, r = r(p) = r(\tilde{p}), i = \{0, .., \frac{h(p,\tilde{p})}{h_p}\}, j = \{0, .., \frac{h(p,\tilde{p})}{h_{\tilde{p}}}\}$:

$$\boldsymbol{\tau}_r \leq \mathbf{s}_{\tilde{p}} + \tau_{\tilde{p}} + j \cdot h_{\tilde{p}} - (\mathbf{s}_p + i \cdot h_p) + h(p, \tilde{p}) \cdot (1 - \bar{\mathbf{x}}_{(p,\tilde{p})}) \tag{4.34}$$

$$\boldsymbol{\tau}_r \leq \mathbf{s}_p + \tau_p + i \cdot h_p - (\mathbf{s}_{\tilde{p}} + j \cdot h_{\tilde{p}}) + h(p, \tilde{p}) \cdot \bar{\mathbf{x}}_{(p,\tilde{p})} \tag{4.35}$$

The constraints define a lower bound for the idle time between the termination of a process and the start of another process on the same resource. The switching variable $\bar{\mathbf{x}}_{(p,\tilde{p})}$ ensures that only one of the constraints is active while the other constraint is trivially satisfied. Based on the lower bound for each resource, we determine the following objective.

$$\max\left(\sum_{r \in R} \boldsymbol{\tau}_r\right) \tag{4.36}$$

The objective maximizes the sum of all lower bounds, thus, introducing a minimal idle time between all processes on one resource.

4.4.3 Maximize Cumulative Idle Time in Schedule

The metrics proposed in the previous subsections were concerned with spreading processes either within an application or single resources. In addition, this subsection proposes a metric to maximize the idle time within the whole cluster schedule, i.e., to generate a long idle time segment over all resources. To implement this metric, we introduce a phantom process p_{idle} representing the idle segment with the following properties.

- $h_{p_{\text{idle}}}$ - period of p_{idle} which equals the smallest process period in the cluster $h_{p_{\text{idle}}} = \underset{p \in P_d}{\text{argmin}}(h_p)$.
- $\mathbf{s}_{p_{\text{idle}}} \in \mathbb{R}$ - start-time of idle segment.

- $\boldsymbol{\tau}_{p_{\text{idle}}} \in \mathbb{R}$ - variable representing the duration of the idle segment.

Figure 4.10(c) illustrates the metric measuring the maximal available idle segment. We can then define the following additional constraints for the ILP formulation (see Section 3.2.2).

$$0 \leq \boldsymbol{\tau}_{p_{\text{idle}}} \tag{4.37}$$

$\forall p \in P_d, i = \{0, .., \frac{2 \cdot h(p, p_{\text{idle}})}{h_p} - 1\}, j = \{0, .., \frac{2 \cdot h(p, p_{\text{idle}})}{h_{p_{\text{idle}}}} - 1\}$:

$$i \cdot h_p + \mathbf{s}_p + \tau_p \leq j \cdot h_{p_{\text{idle}}} + \mathbf{s}_{p_{\text{idle}}} + h(p, p_{\text{idle}}) \cdot (1 - \bar{\mathbf{x}}_{(p, p_{\text{idle}})}) \tag{4.38}$$

$$j \cdot h_{p_{\text{idle}}} + \mathbf{s}_{p_{\text{idle}}} + \boldsymbol{\tau}_{p_{\text{idle}}} \leq i \cdot h_p + \mathbf{s}_p + h(p, p_{\text{idle}}) \cdot \bar{\mathbf{x}}_{(p, p_{\text{idle}})} \tag{4.39}$$

Constraint (4.37) first ensures that the idle segment has a positive value. Constraints (4.38) and (4.39) then ensure that no processes are scheduled in the idle segment. Based on these constraints we define the following objective:

$$\max\left(\boldsymbol{\tau}_{p_{\text{idle}}}\right) \tag{4.40}$$

The objective maximizes the duration of the idle segment, generating long idle segments to accommodate other cluster schedules.

4.5 Experimental Results

This Section presents several experimental results, evaluating our schedule integration framework. First, a scalability analysis comparing our framework with an ILP approach which generates schedules from the scratch is presented. Second, we analyze the feasibility of the schedule integration if a minimal end-to-end delay optimization is applied, and discuss drawbacks of schedule integration. Finally, an evaluation of the metrics for optimizing the cluster schedules is presented. The schedule synthesis has been carried out on an Intel i7 3.4GHz with 16GB RAM. We use Microsoft's Z3 version 4.3.0 as SMT solver for our framework [MB08]. For the schedule integration, we consider each application as an individual cluster independent of the initial cluster size. The ILP approach is based on the formulation in Section 3.2. We use CPLEX in version 12.6 as ILP solver [ILO]. Note that the schedule is obtained at design time such that runtimes of several minutes are still acceptable.

4.5.1 Scalability Analysis for Schedule Integration

To evaluate the scalability of our framework, we compare its runtime with the runtime of the ILP from Section 3.2. For our framework we also include the runtime for generating subsystem

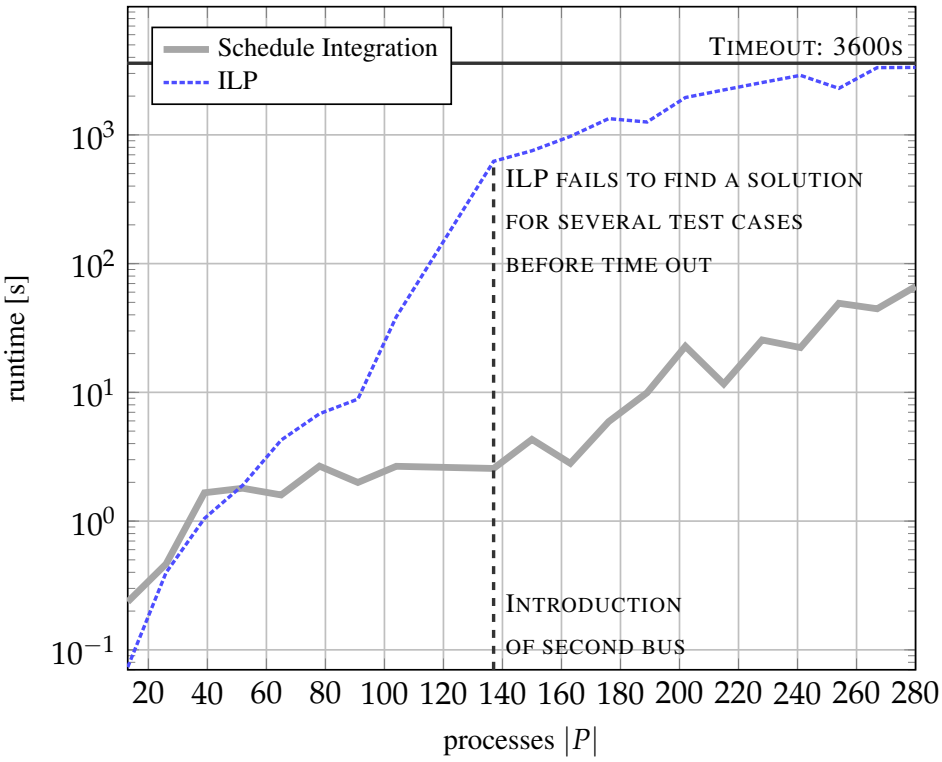


Figure 4.11: *Runtime comparison of our Schedule Integration framework with an ILP approach for* 1000 *synthetic testcases. To improve legibility, the test cases are grouped by their average process number.*

schedules. We have generated 1000 synthetic test cases with different properties. Starting from a small architecture, consisting of three ECUs connected by an Ethernet bus, we add randomly generated applications with varying periods and task execution-times to the system[3]. The utilization of the ECUs is about 50 %. For each 15 tasks an additional ECU is added to the system. Once the system consists of more than 10 ECUs, we introduce a second domain that also communicates through an Ethernet bus. The ECUs are evenly distributed between the domains and one ECU serves as central gateway. For this case study, we have selected a cluster size of 5 applications. These highly heterogeneous test cases allow to evaluate the influence of various properties, such as the number of tasks, the number of resources and domains. For this scalability analysis we omit concurrent message transmission and assume only one message might be transmitted on a bus at a time instant.

For the scalability analysis, we have not defined an optimization metric for the cluster schedule as well as the ILP approach, thus, the schedule synthesis terminates once the first feasible solution is found. For the schedule integration, we have selected a flexible end-to-end delay.

[3]The application task graphs are generated using OPENDSE [Ope] which uses an implementation of Task Graphs For Free (TGFF) [DRW98] to generate applications.

Hence, the schedule integration might increase the end-to-end delay of the initial cluster schedule. Figure 4.11 illustrates the results of the scalability analysis. The results show that for small test cases the schedule integration is outperformed by the ILP approach, as schedule integration introduces a certain overhead compared to generating a schedule from the scratch. However, with an increasing problem size the ILP is unable to find a solution for several test cases within 1 hour, while our framework always finds a solution in less than 70 seconds in average. In particular, the introduction of the second domain leads to a clear increase in the runtime of the ILP, while the additional domain does not have a significant effect on the schedule integration. The reason for this runtime increase is the higher complexity, as all clusters containing processes mapped to multiple domains share the same gateway. Thus, the scheduling problem is stronger constrained compared to a single domain architecture. Schedule integration handles these strongly constrained problems better than the ILP. For 4.9 % of the test cases a conflict refinement was needed for the schedule integration.

The results of this scalability analysis clearly show the potential of schedule integration to efficiently address the problem of scalability for holistic scheduling of time-triggered systems.

4.5.2 Feasibility Analysis for Schedule Integration

The scalability analysis has shown the advantages of the divide-and-conquer approach applied for schedule integration. However, as cluster schedules are generated individually, in several cases it might be necessary to apply a conflict refinement which adapts the initial cluster schedule. In particular, if the end-to-end delay is fixed, schedule integration might be unable to solve schedule synthesis problems that can be solved if the system schedule is generated from the scratch. To analyze this drawback, in the following, we perform a feasibility analysis. The case study consists of the same 1000 synthetic test cases as for the scalability analysis. However, here we consider each application as an individual cluster. For the feasibility analysis, we have defined the following objective minimizing the end-to-end delay:
$\forall a \in A_d, \phi \in \Phi(E_a)$:

$$\theta_a \geq \sum_{p \in \phi} \tau_p + \sum_{(p,\tilde{p}) \in \phi} \mathbf{w}_{(p,\tilde{p})} \tag{4.41}$$

$$min\left(\sum_{a \in A_d} \theta_a \right) \tag{4.42}$$

The objective minimizes the overall end-to-end delay of all applications. It is applied for generating both the cluster schedules as well as to generate a schedule from the scratch (see Section 3.2). Thus, for our framework we generate a schedule with a minimal end-to-end delay for each application individually and try to integrate these schedules into a global schedule. The end-to-end delay determined for the cluster schedule is fixed and might not be extended during conflict refinement.

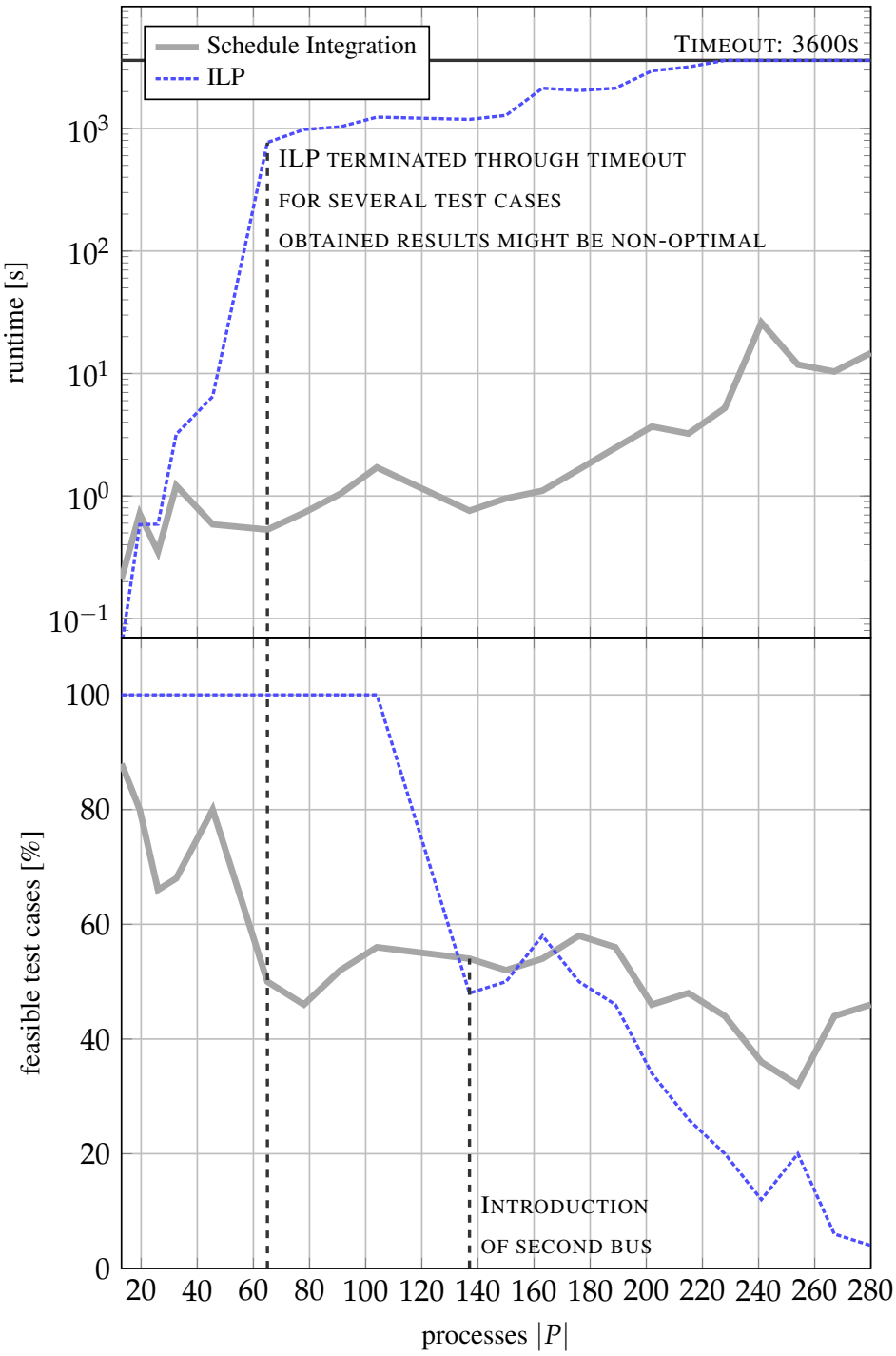


Figure 4.12: *Analysis of runtime and feasibility of our Schedule Integration framework with an ILP approach if an objective to minimize the end-to-end delay is defined.*

Figure 4.12 shows the results for the feasibility analysis. The runtime analysis shows that the ILP reaches the time-out of 1 hour already for test cases consisting of 65 processes. While the ILP is able to determine a feasible solution within this time-frame, the obtained solutions might not be optimal. By contrast, schedule integration terminates the schedule synthesis in less than 30 seconds in average for all problem sizes. This is due to the limited flexibility during conflict refinement.

Analyzing the feasibility in Figure 4.12 shows that the schedule integration is only able to find a solution for $46\,\% - 88\,\%$ of the test cases if the problem only consist of one domain (up to about 100 processes), while the ILP approach is able to solve 100 % of these test cases. However, with an increasing problem size and in particular through the introduction of the second domain, also for the ILP the feasibility is clearly reduced and it is unable to find a feasible solution for several test cases within the given time frame. For large test-cases our framework is able to find a solution for various test cases which the ILP is unable to solve within the time frame of 1 hour. For this cases study, our framework applies a conflict refinement for more than 38 % of the test cases.

Note that the number of feasible test cases generally decreases for the schedule integration with a rising number of processes. However, as various parameters are adjusted concurrently during test case generation, the complexity might differ for some of the test cases. For instance, the deviations for 45 processes and 78 processes are related to how certain applications are mapped to the resources.

This case study illustrates the drawbacks of our schedule integration framework. If strict constraints are defined for each cluster schedule, schedule integration might be unable to find a feasible solution, while generating a schedule from the scratch is feasible given a sufficiently long time frame.

4.5.3 Feasibility Analysis for Integrability Metrics

This section analyzes the influence of the cluster structure on schedule integration by evaluating the metrics proposed in Section 4.4. To evaluate the different metrics, we have generated 1750 synthetic test-cases using the same test case generator as for the scalability analysis. To analyze the influence of different parameters, for the test cases we alternate not only the number of processes, but also the maximal utilization of each ECU and the number of processes per ECU. For this case study, we consider each application as an individual cluster.

To evaluate the integrability metrics, we compare them also with an objective determining a minimal end-to-end delay as proposed in the feasibility analysis. We have defined the following objective functions for the ILP to evaluate their influence on the feasibility of test cases:

(1) Minimize the end-to-end delay (see Section 4.5.2).

(2) Maximize the slack usage (see Section 4.4.1).

(3) Optimize the idle time distribution on each resource (see Section 4.4.2).

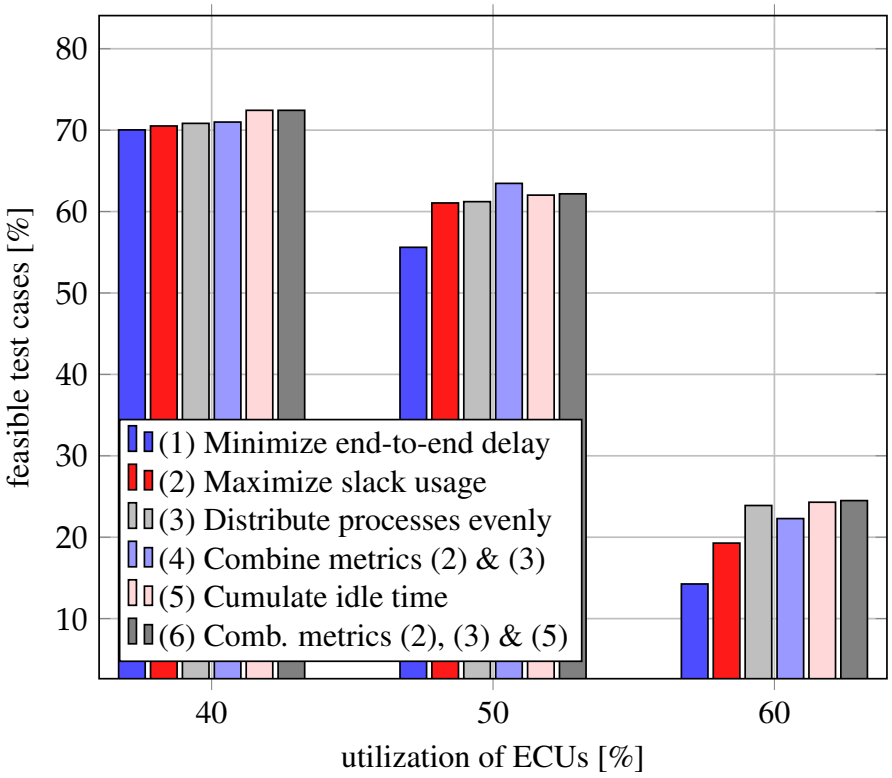


Figure 4.13: *Feasibility analysis of schedule integration depending on the metrics optimized during cluster schedule generation for a different utilizations of ECUs.*

(4) Objective combining (2) and (3), aiming at an increased variability.

(5) Optimize cumulation of idle time over all resources (see Section 4.4.3).

(6) Objective combining (2), (3) and (5), increasing the variability and the general cluster integration.

For the objectives combining several metrics, we have selected the same weight for each metric. For this case study the end-to-end delay for each cluster schedule is fixed, thus, might not be increased during conflict refinement.

Figure 4.13 shows the percentage of feasible test cases depending on the utilization of ECUs. The results show that with an increasing utilization, the number of feasible test cases decreases for all metrics. It can also be seen that objective (1), minimizing the end-to-end delay, is outperformed by all other objectives. It suffers in particular from the increased complexity of a higher utilization, as the tight end-to-end delay bounds limit the variability. Hence, objective (2), maximizing the slack usage, shows clearly better results as the variability to integrate schedules is increased. However, (2) is outperformed by the remaining objectives. Objective (3), evenly distributing processes, shows especially for a high utilization a good performance, as integrating cluster schedules is facilitated through the even distribution of the processes. Objective (4),

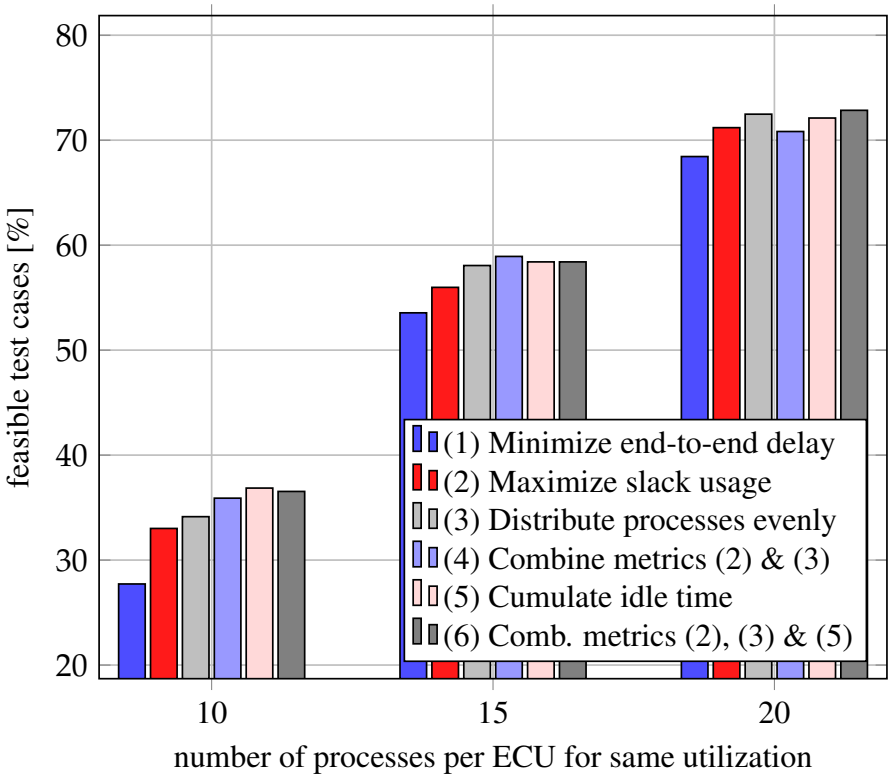


Figure 4.14: *Feasibility analysis of Schedule Integration depending on the metrics optimized during cluster schedule generation for different numbers of processes on each ECU. Note that the utilization of ECUs is the same for all process numbers.*

combining (2) and (3), performs particularly well for a utilization of 50 % but suffers from the poor performance of (2) for a higher utilization. The good performance for 50 % utilization can be explained by the facilitated cluster integration through the high variability of the cluster schedules. Overall, objective (5), introducing a maximal idle time over all resources, and (6), combining all integrability metrics, show the best performance of all metrics. Optimizing the idle time is especially beneficial for problems with a low utilization, as the created idle times accommodate additional clusters well. However, also for a high utilization optimizing the idle time is beneficial, as the idle segments facilitate the integration and aid the conflict refinement. Objective (6) overall leads to the highest number of feasible test cases.

In addition, Figure 4.14 evaluates the overall feasibility for a varying number of processes per ECU for the same 1750 test cases. The results show that for the same utilization a low number of processes with a high execution-time leads to a lower feasibility compared with a high number of processes with a low execution-time. While these results might seem counterintuitive, they verify the outcome of the scalability and feasibility analysis, indicating that schedule integration deals well with strongly constrained problems if the flexibility is high, i.e., in this case the number of processes on a resource and their offsets. The analysis of the objective func-

Table 4.1: *Analysis of influence of applied objective function for cluster scheduling on the feasibility of the schedule integration.*

metric	**(1) Min. e2e**	**(2) Max. sl.**	**(3) Ev. dist.**	**(4) (2)&(3)**	**(5) Cum. idle**	**(6) (2),(3)&(5)**
feasible	48.97 %	52.52 %	54.01 %	54.41 %	54.98 %	55.10 %
av. runtime	11.37 s	25.09 s	28.23 s	31.02 s	36.14 s	34.94 s

tions reflects very similar results as for the evaluation based on the utilization. Objective (4) shows the best performance for the test cases with a medium difficulty and metric (6) shows the best overall performance. However, objective (4), combining (2) and (3), is outperformed by both for 20 processes per ECU. Similarly, objective (5) performs slightly better than (6) for 10 processes per ECU. This indicates that fine-tuning the weights for objectives (4) and (6) would allow to improve their results.

Table 4.1 gives an overview of the overall feasibility of all applied objective functions and the average runtime of each approach. The runtime comparison shows that the improved feasibility also leads to an increased runtime for the respective objective function. Note that not all test cases might be feasible as only for 65.17 % of them at least one of the objective functions allowed to find a solution. Thus, the actual percentage of feasible test cases might be higher for all metrics.

These results have shown the general potential of the different integrability metrics, but also indicate the need for an extensive analysis of different parameters to further improve the integrability.

4.6 Design Flow: Enabling Highly Modular Architectures Based on Schedule Integration

The schedule integration framework presented in this chapter addresses the problem of integrating individually designed subsystem schedules into a global schedule. This section proposes an extension of our schedule integration framework with a data-centric description to enable highly modular architectures. A data-centric description allows to decouple the design process of individual software components, as functionality can be implemented independently of the underlying hardware architecture. This design approach is in accordance with the AUTOSAR

architecture which aims at abstracting the underlying hardware and provides common interfaces (see Section 1.3.2). In the following, first the subsystem design is discussed in more detail, before a design flow from an individual software component to a fully configured E/E-architecture is presented.

4.6.1 Data-centric Design

A data-centric description specifies application requirements together with interfaces defined by topics. This helps to decouple the development process of individual software components and forms the basis for a modular system that is highly composable [PCFW05]. In a data-centric system, the supported and required properties of each software component are described explicitly. This includes intrinsic properties like the processing time, memory footprint, execution period, and security level as well as extrinsic properties like the required and provided data with quality-of-service descriptions or end-to-end delay requirements. Either a design tool or an appropriate middleware is responsible to find a suitable matching between the different property sets and to calculate a configuration for the whole system, including schedules and routing of messages. The benefit of this approach is that developers can focus on the properties of local software components and do not have to consider the global interaction between them. Each application can have a set of publishers for sending data and a set of subscribers for receiving data. Each publisher/subscriber is associated with a certain topic which exactly defines the *type* of data and a *name*, e.g., *{Steering Wheel Angle}* () might have the type *{degree [rad]}*. The system designer abstracts any hardware-specific functionality, like reading sensor data through an interface defined by an explicit topic. This has the major advantage that hardware components or applications can easily be replaced. Figure 4.15 shows the resulting task graph for a *steer-by-wire* application after the links between the application tasks and the sensor and actuator tasks have been established.

While AUTOSAR does not directly follow a data-centric approach, topics might be realized in a more rigid and simple form using ports. Links and communication paths are then statically configured during the system design.

4.6.2 Data-centric Design Flow

The proposed design flow is illustrated in Figure 4.16. It is based on a data-centric design and a time-triggered architecture. (1) The application developer implements software components independently of the hardware platform of deployment. Interfaces to other applications are defined by *topics*, clearly specifying the required or provided data. (2) To integrate a data-centric application, links between publishers (e.g. sensor task) and subscribers (e.g. computation task) are established, obtaining the final task graphs. To access sensor data or control actuators, the system designer implements interfaces which applications might subscribe to depending on their topic. (3) As tasks abstracting hardware peripherals are explicitly bound to a particular

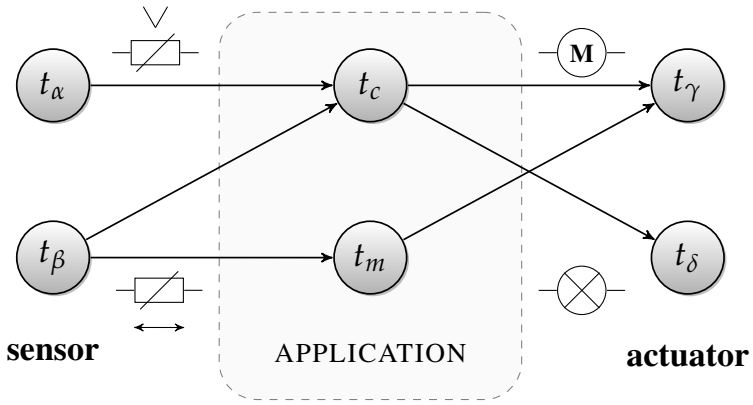


Figure 4.15: *Exemplary illustration of a data-centric steer-by-wire application. The application subscribes to two sensor topics (⇙,⇙) to calculate the steering angle published to the actuator topic (⊙). An indicator topic (⊗) allows indicating errors. The application tasks t_c and t_m only have data but no hardware dependencies.*

hardware resource, an implicit task mapping is given. For applications distributed over several network nodes, additional messages are introduced if required. The resulting specification defines all process properties and resource mappings required to apply a schedule synthesis. (4) An application schedule is created, specifying the exact starting times for all tasks and messages. For this schedule synthesis, the approach presented in Section 3.2 is applied. Depending on the application requirements, an optimization objective might be defined for a minimal end-to-end delay, the control performance [GLSC12], or the integrability to ease the subsequent schedule integration (Section 4.4). (5) The schedule integration approach presented in this chapter is applied to integrate the individual subsystem schedules into a global schedule. It allows to add or update application or subsystem schedules in an iterative process, measurably reducing the integration efforts.

AUTOSAR allows to apply this design-flow for the design of current automotive E/E-architectures, as it provides a homogeneous software platform and abstracts the communication between participants for the application. A data-centric description might therefore extend the current AUTOSAR interfaces and ports. After our framework was applied to synthesize the system schedule, a data-centric design-tool might then generate AUTOSAR configuration files defining the communication paths and schedules. This would allow to integrate the proposed design flow using schedule integration in an AUTOSAR-based system design with the goal of a static configuration.

In addition, with an increasing number of software functions, updating and maintaining software in the field will gain importance in the next years. Furthermore, the introduction of an extended in-vehicle appstore, allowing to update and install also safety-critical functionality, is a likely scenario. The proposed data-centric design flow would then allow to efficiently integrate

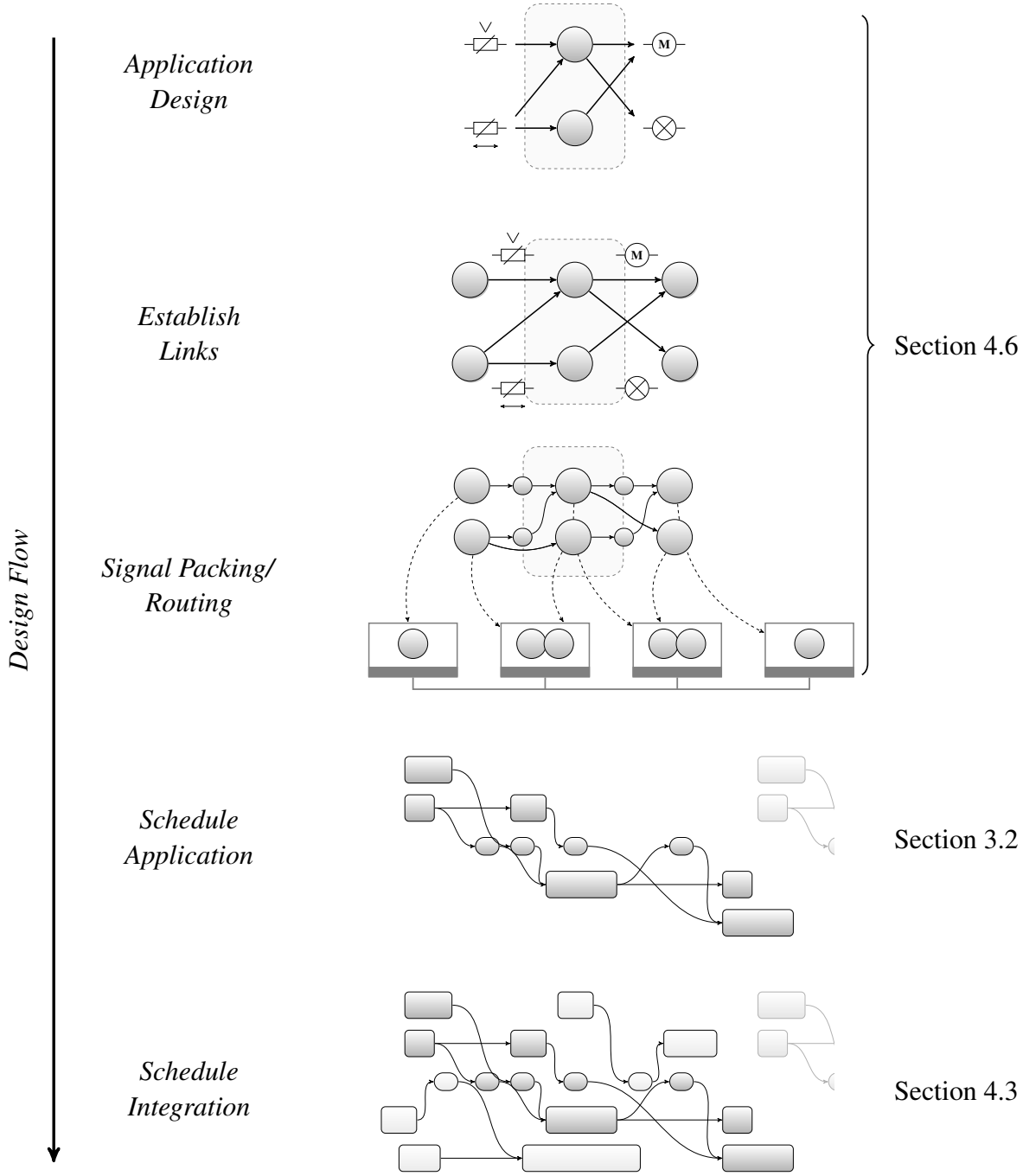


Figure 4.16: *Design flow from a data-centric application definition to a fully configured system. Schedule integration is applied to integrate the individual application schedules in a global system schedule.*

new functionality in the system, giving real-time guarantees if a valid configuration can be found. The basis for such a flexible architecture forms a data-centric middleware. For instance, the middleware *SOME/IP* based on Ethernet developed by BMW [Völ13], or the middleware *CHROMOSOME* [BGG+14] already provide publish-and-subscribe mechanisms at runtime and might form the basis for a flexible architecture where applications can be easily added and updated.

4.7 Summary

In this chapter we presented a schedule integration framework enabling a modular schedule synthesis. Schedule integration allows to integrate independently developed application or subsystem schedules into a global system schedule. We proposed a multi-stage approach which first applies an SMT-based integration of subsystem schedules without altering the initial structure. If this basic integration fails, we apply a conflict refinement which adapts conflicting subsystem schedules if required. A partitioning metric allows to reduce the complexity of the conflict refinement by defining suitable subproblems that might be solved individually. The framework supports holistic scheduling for E/E-architectures communicating with Automotive Ethernet and FlexRay. In addition, three integrability metrics to optimize subsystem schedules for schedule integration were presented.

The results, consisting of a scalability and a feasibility analysis showed the benefits of schedule integration. In particular, the very good scalability of schedule integration in comparison to a conventional ILP approach generating the schedule from the scratch was shown. This allows to apply our schedule integration framework also to large problems where conventional schedule synthesis approaches become intractable. Furthermore, the results show that schedule synthesis is suitable for a modular design approach where subsystem schedules can be generated independent of each other.

In this chapter, we assumed that all subsystem schedules were developed completely independent of each other, i.e., conflicting cluster schedules are likely. For instance, the results indicated that fixing the end-to-end delay for subsystem schedules might reduce the feasibility of schedule integration. However, the results also showed that for large test cases this feasibility still noticeably exceeds the results obtained by the ILP if a maximal runtime is defined. To improve the integrability of clusters, we therefore proposed several integrability metrics. The results showed that the metrics are suitable to improve the integrability of cluster schedules and help to increase the applicability of schedule integration. Optimizing the weights for each metric if several are combined to one objective, might allow to further improve these results.

In summary, our schedule integration framework has shown that it (1) allows to clearly improve the scalability of mathematical approaches for holistic scheduling, and (2) enables a modular schedule synthesis in accordance with the design approach of automotive systems. Schedule integration is therefore well-suited to enable highly composable architectures as envisioned by AUTOSAR. To further improve its applicability, an important line of future work is

an extensive analysis selecting suitable weights for the integrability metrics and to incorporate available knowledge about other clusters in the cluster schedule generation.

In the following chapter, a variant-aware schedule synthesis is proposed which relies on schedule integration to improve the scalability. We also present a partitioning heuristic which allows to select suitable clusters.

5

Schedule Synthesis for Variant Management

The previous chapter has investigated a modular schedule synthesis using schedule integration. In this chapter we investigate another aspect of schedule synthesis, generating schedules for multiple system variants. In this context, schedule integration is applied to improve the scalability of the variant-aware schedule synthesis presented in the following.

Today, *variant management* is one of the key challenges that car manufacturers face. In the automotive industry, all mass production manufacturers provide their customers with various car models and a large variety of customization options. For instance, AUDI currently offers 49 different car models and this number will increase to 60 by 2020, each offering hundreds of configuration options [Cap13]. Handling all these variants is a significant challenge both during design and manufacturing. Efficient variant management not only reduces development cost, it also allows to manufacture different models at one assembly line and is therefore a significant economical and competitive factor. While concepts like the *Modular Transverse Matrix* of Volkswagen Group exist in industry [Lup12], techniques for variant management of E/E-architectures have not been systematically studied so far.

This chapter addresses the problem of a variant-aware schedule synthesis. We assume a time-triggered architecture as presented in Chapter 2, applying synchronous time-triggered scheduling. Figure 5.1 illustrates the architectures of three vehicle variants, e.g., versions for a petrol, a diesel and a battery electric vehicle. All three variants share common applications, e.g., Anti-lock Braking System (ABS), but also have exclusive applications such as Adaptive Cruise Control, Lane Assist or the motor control for different engine types. To generate system configurations for these variants, the manufacturer has three options. (1) Determining an independent configuration and schedule for each variant, (2) defining a single global schedule for all vari-

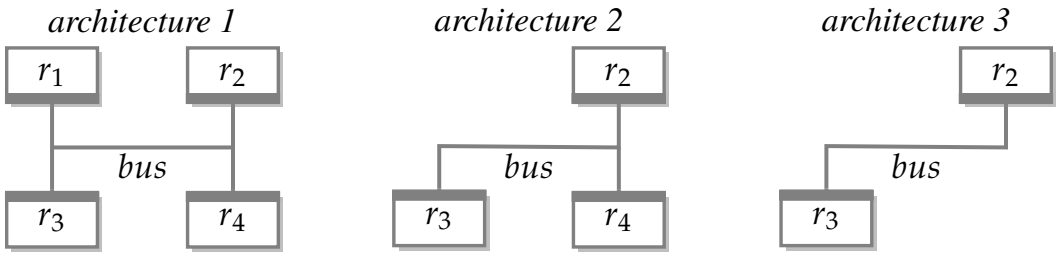


Figure 5.1: *Hardware architectures of three different variants, consisting of a different number of ECUs connected over a bus.*

ants, and (3) *multi-schedule* synthesis, as proposed here. Option (1) suffers from a significantly increased testing and integration effort as common applications are not variant-independent and have to be tested individually for each variant. As opposed to this, options (2) and (3) allow to test application configurations independently of the variant they are deployed in and common applications have to be tested only once. Option (2) leads to a significant overestimation of resource requirements as exclusive applications, like the motor control for a petrol and a diesel engine, considered in the global schedule, will not be deployed in the same variant and reserved resources remain unused. By contrast, option (3) generates an individual schedule for each variant but contains an identical schedule for tasks and messages common to multiple variants. This leads to an efficient usage of available resources, while the testing and integration efforts are reduced. For instance, the configuration of a diagnosis tool might be reused for all variants, or the integration of ECUs is facilitated as the configuration has to be done only once [DH14].

The significance of a variant-aware schedule synthesis becomes apparent when looking at the design process of automotive architectures. The basic system schedule is defined at an early design stage and the applications are then developed independently. Here, our multi-schedule synthesis can particularly reduce the impact of a cost-intensive and error-prone *integration testing*, which verifies the function, performance and reliability of the entire system [SN13], as each application is variant-independent and can be tested individually. During the integration, the time-triggered multi-schedule then ensures that applications do not interfere in different variants. As a result, the testing and integration effort is reduced, leading to a reduction of the overall design time of the vehicle.

Methodology. Figure 5.2 illustrates the task graphs of the three vehicle variants from Figure 5.1, sharing common tasks and messages but differing in the underlying architecture. We propose a schedule synthesis approach generating a time-triggered multi-schedule that defines release-times for periodic tasks and messages, as illustrated in Figure 5.3. Our framework determines common parts in multiple variants and defines an identical schedule for this commonality. For instance, for m_1 and t_2, common to all variants, the same schedule is assigned. As the commonality between all variant specifications might be low, we propose an incremental approach

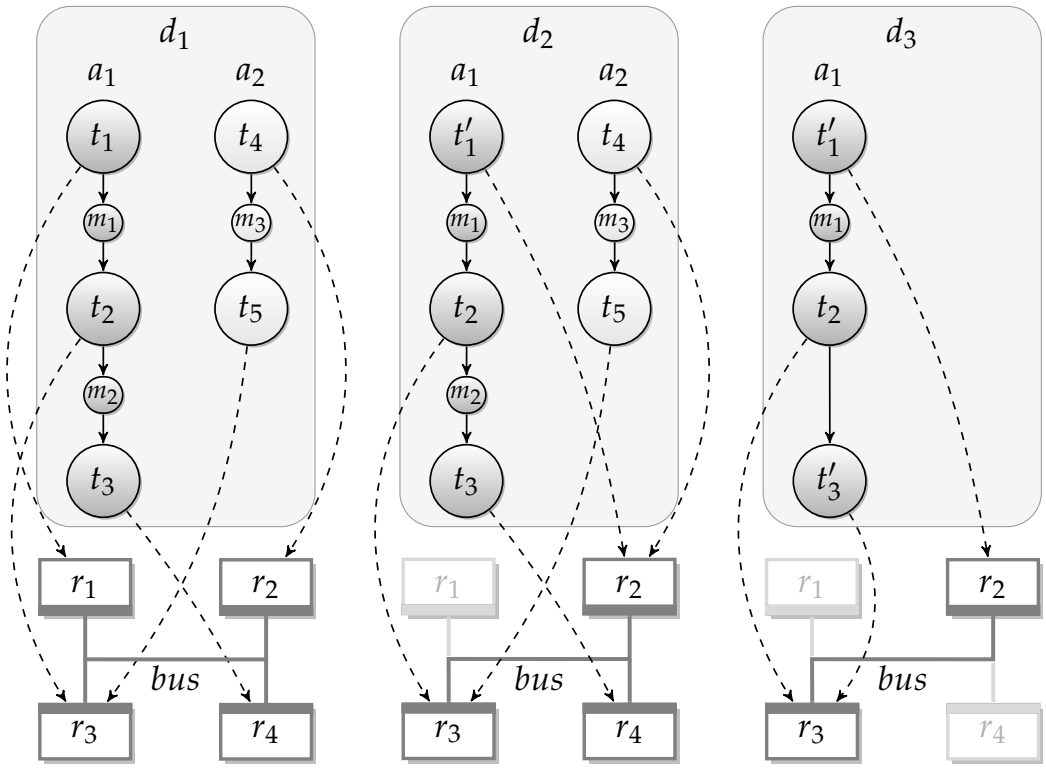


Figure 5.2: *Task graphs including a task to resource mapping for three variants. Variant specification d_1 describes the task graph of the full system using four ECUs, while d_2 and d_3 are variants utilizing three and two ECUs, respectively. For instance, d_3 only supports application a_1, but not a_2.*

which also considers commonality in a subset of variants, e.g., t_4, m_3, and t_5 shared by variants d_1 and d_2.

Related work. The problem of defining multiple configurations for a system has been addressed in the area of *multi-mode* scheduling, however, with a different objective. While multi-mode approaches focus on optimizing the switching of the system configuration of a single system, often with the goal of minimizing the transmission delays between modes, e.g., [NNS+11, ACPF14], we address the problem of determining variant schedules with minimal differences for multiple architectures. Furthermore, several approaches have been proposed for concurrent scheduling of multiple graph-based applications, e.g., [IO98, ZS06]. While these approaches address a similar problem of merging different applications into a single task graph, the goal is to generate a single schedule instead of multiple variant schedules. Furthermore, despite various tools being available to assist in the variant management in the automotive domain [pur06, Vec08, Men10], the problem of a comprehensive and holistic variant management still remains open. Despite its importance for the industry, variant management has gained only little attention in the scientific community. For instance, in [BD07] a graph-based representation to enable the use of graph

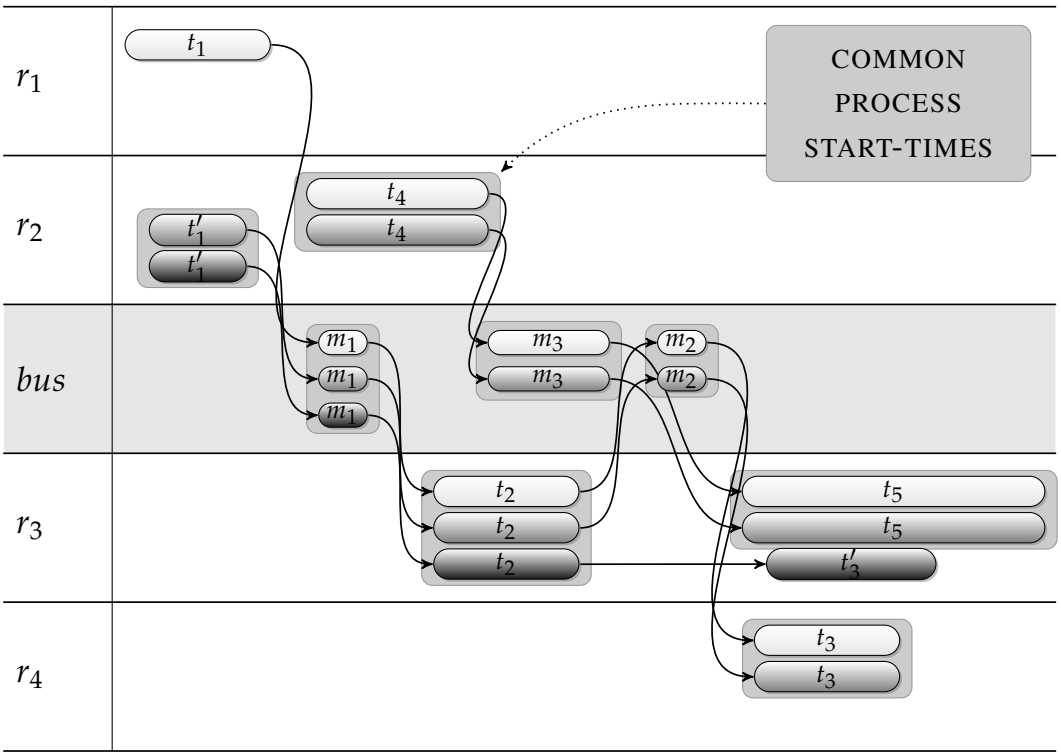


Figure 5.3: *A multi-schedule defines the same schedule for tasks and messages shared by the three variants d_1,d_2,d_3 defined in Figure 5.2*

theoretic analysis tools is presented, while [GGTL14] proposes a multi-variant based DSE. Finally, a variant-aware schedule synthesis was first addressed in [DH14]. The authors propose an approach specific to message scheduling for the FlexRay bus. By contrast, the work at hand proposes a holistic approach, taking both tasks and messages into account during schedule synthesis. A detailed survey of existing work is given in Section 5.1.

Contributions. This chapter proposes a framework for multi-schedule synthesis for variant management. Multi-schedule synthesis involves several challenges, viz., the determination of common parts in a set of variant schedules, the generation of a common schedule for these parts, and the integration of uncommon parts. In addition, the complexity of the schedule synthesis is clearly increased, as multiple variant schedules have to be scheduled concurrently. We therefore propose an incremental approach which first determines a multi-schedule for common parts in all variants, before addressing commonality in variant subsets, extending the multi-schedule. A graph-based approach determines common parts in variants and an SMT approach is applied for schedule synthesis. To improve the scalability, we also propose a partitioning heuristic using graph-based metrics to divide the schedule synthesis problem into smaller problems. This approach is in accordance with the organization of automotive architectures in domains, i.e., partitions, corresponding to the functionality such as chassis or drivetrain applications (see Sec-

tion 1.1). Schedule integration (see Chapter 4) is applied to integrate the partition schedules in a common schedule. The focus here lies on non-preemptive time-triggered scheduling as we exploit the temporal composability of time-triggered architectures [KB03, ATD+09] for the incremental extension and the partitioning. Our framework does not aim at obtaining globally optimal schedules, but rather at determining variant schedules with minimal differences, satisfying predefined maximum end-to-end delays. Multi-schedule synthesis minimizes the differences between different variants and therefore reduces the testing and integration efforts, as it only has to be done once, while using available resources efficiently.

In summary, our contributions are, (1) an incremental multi-schedule approach generating time-triggered variant schedules that consider commonality, and (2) a heuristic partitioning of the schedule synthesis problem which enables a divide-and-conquer approach to significantly reduce the runtime.

Outline. The rest of the chapter is structured as follows. The next section discusses related work. Section 5.2 gives a high-level introduction to our framework. Section 5.3 presents implementation details of our multi-schedule synthesis and the partitioning metric. Finally, Section 5.4 evaluates our framework with three case studies and an automotive lab setup.

5.1 Related Work

In this chapter, we address the problem of multi-schedule synthesis, generating individual variant schedules while assigning the same release-times to shared tasks and messages. We would like to distinguish our approach from work which has used the terms *multi-schedule generation* and *multi-schedule synthesis* in a different way. For instance, [DP94] proposes a "multi-schedule approach" in the context of digital circuit design, where the problem of a premature pruning of the design space using a single schedule is overcome by generating and synthesizing multiple optimal schedules. However, as that approach tries to find an optimal solution for one given specification, it is not comparable with our work.

Variant management for automotive E/E-architectures is highly important for the industry, being reflected in the increasing number of industrial tools. The growing product complexity and model diversity give variant management a prominent position within the software development process where features describe the commonalities and variabilities of a product line [pur06, Men10] and complex calibration processes are structured [Vec08]. Although, currently, these commercial tools do not address the schedule synthesis problem of time-triggered schedules, our techniques could be incorporated into such tools in the future. In [BD07] the authors propose to transfer variants from the multiple-domain matrix representation into a graph representation in order to apply graph theoretic analysis tools to variant management. In the context of E/E-architectures, our framework follows this approach by using graph-based specification models to generate variant schedules. One recently published approach proposes a multi-variant-based DSE [GGTL14]. Based on a 0-1 ILP, the authors develop architectures for

all defined variants as well as the overall architecture selection, called *Baukasten*, but do not address the schedule synthesis problem itself. Our approach, on the other hand, assumes that the architectures are given and applies the multi-schedule synthesis on them. Hence, as both works cover different stages of the E/E-architecture design, we cannot directly compare them. Nevertheless, the system models of functional variants might be used as input for our multi-schedule framework in order to generate suitable variant schedules. To this effect, the approach in [GGTL14] can be regarded as a form of preparatory work for our multi-schedule synthesis.

Several approaches have been proposed to improve the *extensibility* of E/E-architectures, hence, minimizing changes necessary to add additional functionality or update the current software. For instance, [ZYN+10] presents a task allocation and priority assignment approach for event-triggered scheduling which optimizes the system based on potential changes, i.e., the increase of task execution-times. Similarly, [EB10] presents a task allocation approach which minimizes the changes required in the future, e.g., changing a task priority, based on potential change scenarios. An upgrading algorithm allows to extend the initial system configuration with a minimal number of changes. While defining different variants as scenarios would theoretically make these approaches applicable for variant-management, the approaches aim at optimizing a single system instead of finding an optimal solution for multiple system variants. Consequently, approaches optimizing the extensibility of a single system suffer from an inferior resource utilization compared to a variant-aware approach optimizing multiple variants concurrently.

In the area of *hierarchical scheduling* for *component-based* systems [DB05, SL08, KWL+12] and the related area of *partition scheduling* [LKY+00, TSP11] various research has been carried out on modular systems. The basic idea is to partition the available resources into suitable runtime budgets or partitions which are then exclusively assigned to a subsystem. If a time-triggered partition scheduler is applied, the resource is then partitioned into time-windows during which one subsystem has exclusive access to the resource. In each component or partition an individual scheduler is responsible to determine the task execution. Thus, the component abstracts the resource requirements of the underlying tasks and the approaches are concerned with determining suitable timing requirements for each component. This would make this body of work theoretically applicable for a variant-aware schedule synthesis, i.e., by defining an upper bound for the timing requirements of different component variants. Based on these upper bound requirements a single global schedule might then be created in which the variant specific components are deployed. However, similar to a global schedule where the tasks and messages of all variants are scheduled in a single schedule, this would lead to an overestimation of the resource requirements. Instead, if approaches for component-based design and partition scheduling are used to determine timing requirements for different variants, the proposed multi-schedule synthesis framework might be extended to schedule event-triggered tasks and messages.

In the area of *multi-mode* scheduling the problem of generating multiple configurations for a system has been addressed. Here, the focus generally lies on optimizing the switching of

the system configuration, often with the goal of reducing the mode change transition latency, i.e., the settling time during the switching from one application mode to another, and the corresponding timing constraint guarantees. For instance, in [SPT09] the authors propose a multi-mode timing analysis method for single-processor systems using Fixed Priority (FP) and EDF scheduling, whereas in [NGA09] two mode transition protocols for preemptive FP scheduling on multiprocessors are presented. However, these approaches focus on event-triggered systems and do not explicitly consider a message-based communication and are therefore not comparable with our work. An algorithm to determine an upper bound for mode change transition latencies for communication-based applications is presented in [NNS+11], targeting a static preemptive priority-based scheduling and asynchronous mode change protocols. By contrast, the authors in [ACPF14] use TDMA for state-based scheduling. More precisely, a workflow for generating communication schedules with optimized average mode-change delays is presented. However, the paper addresses the problem of minimizing transition times, while our approach aims at the minimization of differences between variant schedules addressing a very different problem.

Besides this, there exist several approaches dealing with the concurrent scheduling of multiple graph-based applications in a heterogeneous distributed system. In [ZS06] the authors present four methods to merge different Directed Acyclic Graphs (DAGs) into one composite DAG in order to enable the use of any conventional single-graph scheduling algorithm. Thereby, the objective is to minimize the overall makespan (i.e., the total length of the schedule) and achieve fairness in terms of an equal delay for all DAGs due to a shared utilization of resources. In contrast to this, in [IO98] a dynamic DAG scheduling framework for multiple applications in a distributed environment is presented. Here, the scheduling is done in a decentralized manner, with each application making its own scheduling decisions based on the estimated network loads. Although, these approaches might perform well for the actual scheduling of multiple graph-based applications they do not consider possible shared processes among the different DAGs. To the best of our knowledge, our approach is the first one to merge common tasks and messages of different DAGs with the objective to generate individual variant schedules with minimal differences between them.

The proposed multi-schedule framework applies a holistic schedule synthesis for synchronous time-triggered scheduling which has been discussed in detail in the introduction (see Section 1.7) and the previous chapter (see Section 4.1). All these approaches address the problem of generating a single schedule for a system. Our framework can therefore be seen as an extension of these approaches, generating individual variant schedules using a multi-schedule.

Finally, a first approach for a variant-aware schedule synthesis was presented in [DH14]. The paper addresses the problem of multi-schedule synthesis for time-triggered communication on the FlexRay bus. The focus lies on the message transmission, while in the work at hand we apply a holistic approach for concurrent task and message scheduling. A holistic approach introduces various additional challenges, e.g., task and message relations and an increased problem

complexity. Consequently, we have selected a very different technique than [DH14], relying on a graph-based representation and an iterative SMT approach.

5.2 Framework

In the following, we first introduce the multi-scheduling problem formally, before presenting our multi-schedule framework.

5.2.1 Problem Description

This chapter addresses the problem of multi-schedule synthesis for time-triggered systems which determines individual schedules for a set of variants that share the same schedule for common parts. Figure 5.3 (page 116) illustrates a multi-schedule, defining an identical schedule for shared tasks and messages, e.g., for m_1 and t_2 common to all variants. Furthermore, it considers commonality within a subset of variants, e.g., t'_1 shared by variants d_2 and d_3 or t_4, m_3, and t_5 shared by d_1 and d_2. The objective of the multi-schedule synthesis is to minimize the differences between variant schedules while satisfying all maximum end-to-end delays. To determine variants of a single application, we assume that the application designer has already selected a suitable level of granularity, such that common functionality is implemented in a common task rather than being included in different tasks. Due to the distributed nature of automotive E/E-architectures, this is generally the case.

5.2.2 Multi-Schedule Synthesis Framework

Our framework is based on an incremental schedule synthesis. (1) Common tasks and messages are determined. (2) A *comprehensive task graph* is generated, representing an extended task graph containing the processes of all variants with data-dependencies to the common processes. (3) A multi-schedule is determined. These steps are repeated for all variant subsets, extending the multi-schedule.

Figure 5.4 illustrates the first two iterations of the multi-schedule synthesis for the three variants in Figure 5.2 (page 115). We first determine shared processes for all variants as illustrated in Figure 5.4(a). Based on this common subset, we define a comprehensive task graph. A comprehensive task graph contains all processes and their data-dependencies of applications that share processes, e.g., the comprehensive task graph in Figure 5.4(b) not only contains the shared processes m_1 and t_2, but also the exclusive processes t_1 and t'_1, as well as t_3 and t'_3. Hence, a comprehensive task graph represents a set of variants in a single task graph. Based on this representation, we determine a time-triggered multi-schedule as illustrated in Figure 5.4(c). In the second iteration, we extend this multi-schedule with applications shared by d_1 and d_2. We first determine common parts as illustrated in Figure 5.4(d), only taking processes into account which have not been scheduled yet. After a comprehensive task graph was created (Fig-

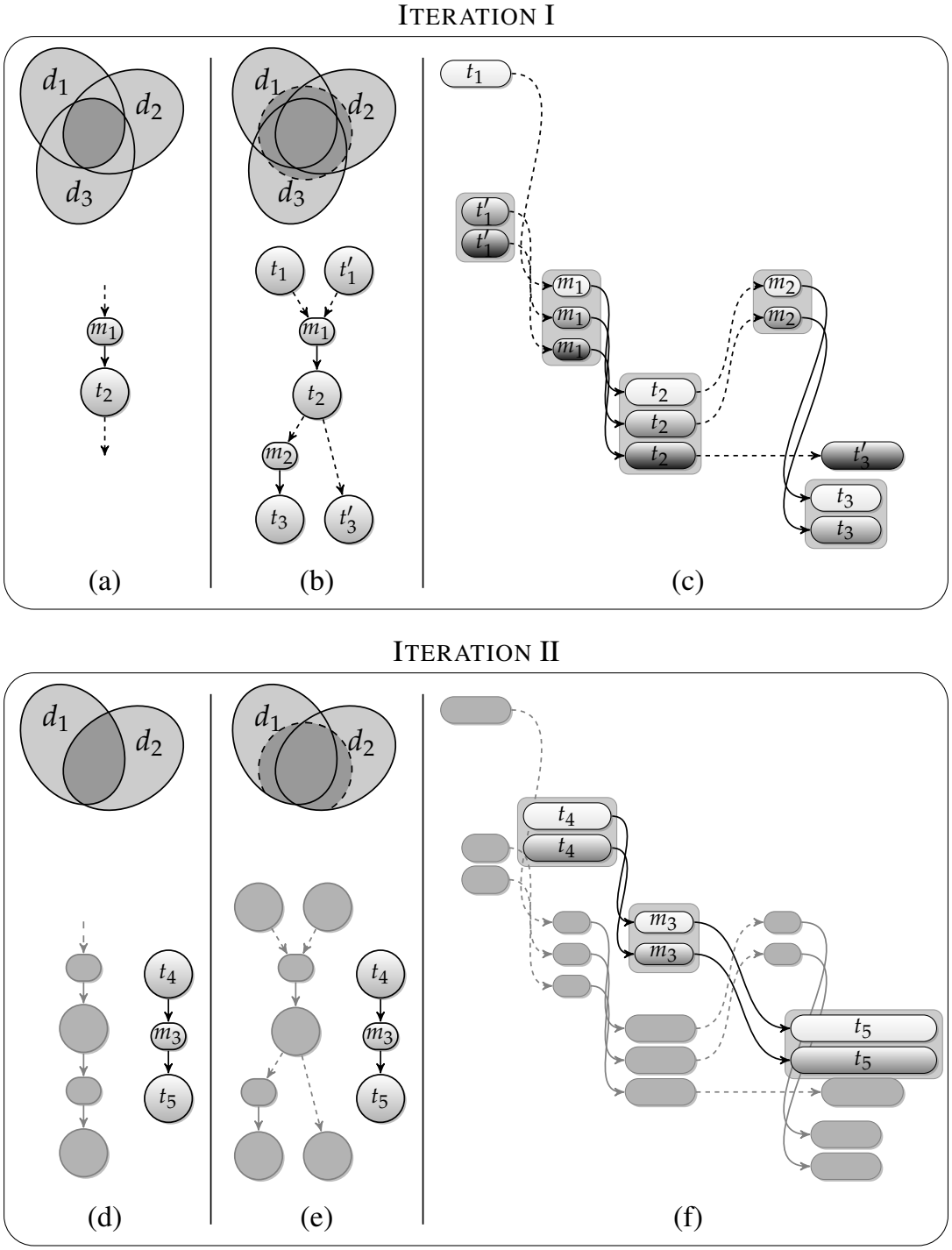


Figure 5.4: *First two iterations of a multi-schedule synthesis for variants from Figure 5.2. Each iteration (1) determines shared processes (a,d), (2) extends these to comprehensive task graphs (b,e), (3) generates a multi-schedule for the comprehensive task graph (c,f). Grayed out elements represent parts which have already been handled in a previous iteration. For schedule synthesis these elements are taken into account when the schedule is generated.*

ure 5.4(e)), we extend our multi-schedule as illustrated in Figure 5.4(f). The process scheduling determined in the previous iteration, illustrated as a grayed out schedule, is taken into account in the form of additional constraints. This iterative process is continued until all processes have been scheduled. Finally, the obtained multi-schedule is converted into individual variant schedules.

Partitioning. Automotive networks are organized in domains, allocating applications within one domain to common ECUs. Hence, applications of different domains only share few resources. To exploit this property, we propose a partitioning-based method to increase the scalability of our framework. Partitioning a task graph and solving the scheduling problem for each generated subgraph is possible as re-integrating the individual solutions to a multi-schedule can be done efficiently using schedule integration as shown in the previous chapter. We therefore propose a partitioning approach to generate suitable subproblems for multi-schedule synthesis. Note that for small subsystems no partitioning might be necessary and the entire system is considered as a single partition.

Framework. The framework iteratively schedules subsets of variants with common parts. For each iteration, it first determines a comprehensive task graph before calculating a multi-schedule. We apply a partitioning heuristic which splits the comprehensive task graph into suitable subgraphs[1]. The schedule synthesis is then applied to each subgraph independently. It calculates a start-time for each process, extending the multi-schedule of previous iterations. As iterative scheduling might lead to conflicts, we also present a conflict refinement. It determines already scheduled applications, causing a conflict, and adds these applications to the currently processed subgraph to adjust the initial schedule, thus, resolving the conflict. After all subgraphs have been scheduled, schedule integration combines the subschedules in each iteration. Once all processes have been scheduled, the multi-schedule is converted into individual variant schedules. Figure 5.5 illustrates the flow chart of our framework. For implementation details and a description of the notation, refer to the indicated sections.

Trade-offs. The goal of multi-schedule synthesis is to minimize the differences between variants, consequently reducing costs through a reduced testing and integration effort. However, while generating individual schedules allows to optimize the application performance for each variant, multi-schedule synthesis might lead to suboptimal performance due to additional constraints for the commonality between variants. Our framework ensures that all end-to-end requirements are fulfilled, thus, guarantees the correct functionality of applications while not guaranteeing optimal performance. This is in accordance with the design approach in the automotive industry where a minimal cost implementation fulfilling all requirements is generally the

[1] Instead of applying the partitioning for each iteration, the multi-schedule synthesis might also be applied for each domain/partition individually before integrating the final multi-schedules into a global multi-schedule. However, as the number of variants generally clearly exceeds the number of domains, to improve scalability, we apply the partitioning of the whole system for each iteration.

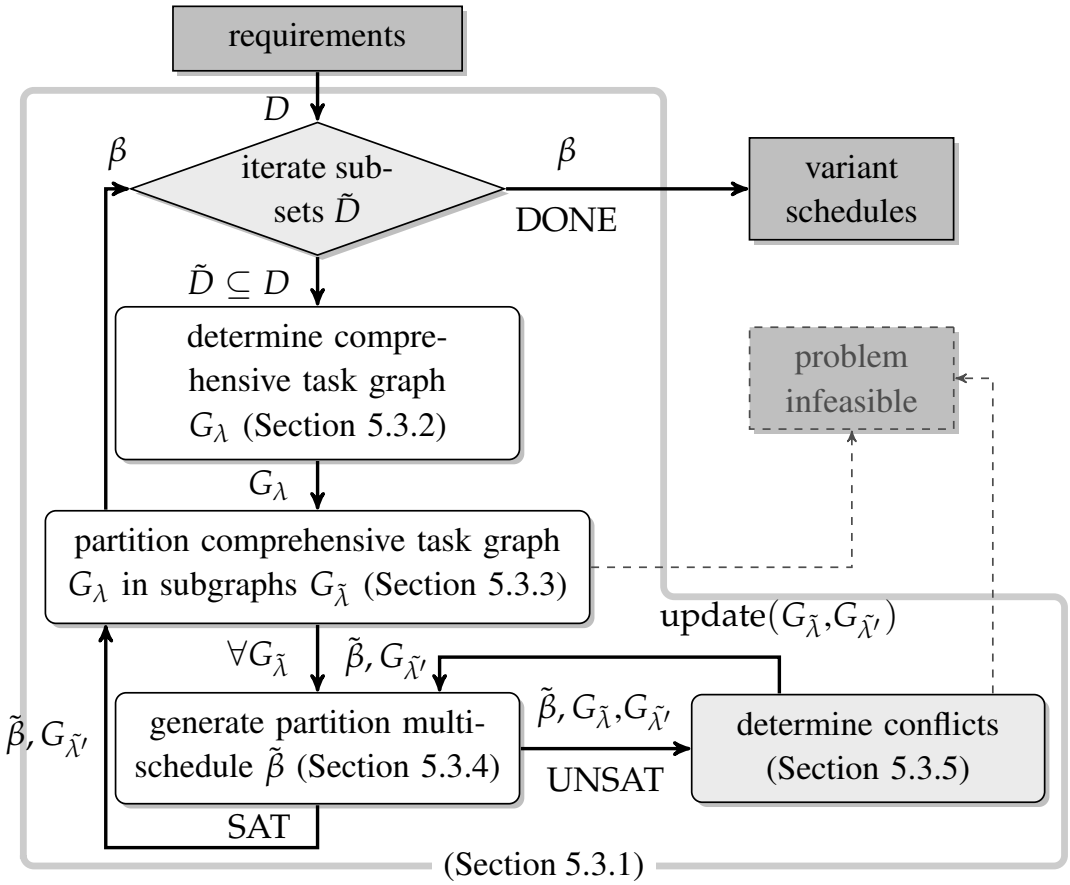


Figure 5.5: *Flow chart of multi-schedule synthesis framework. Individual schedules are iteratively determined for a set of variants. The framework first determines a multi-schedule for shared processes and extends this schedule iteratively with remaining processes. A conflict refinement resolves infeasibilities with schedules from previous iterations.*

goal. However, in some cases, obtaining a multi-schedule might not be feasible while a solution for generating each variant schedule individually is possible. In this case, our framework could provide information about conflicting parameters to the system designer, serving as guidelines on how to adapt the system specification, e.g., through adjusting the number of variants. Note that a single global schedule deployed in all variants not only includes all common parts in the schedule but also the uncommon parts of all variants. Consequently, if a single global schedule exists, also multi-schedule synthesis is applicable but not vice versa.

5.3 Multi-Schedule Synthesis

This section proposes the multi-schedule framework, as illustrated in Figure 5.5. We first introduce the outer-loop of our framework which incrementally iterates over the subsets. Second, we

present an algorithm to determine *comprehensive task graphs*, as introduced in Section 5.2.2. Third, we propose a partitioning approach to improve the scalability of our framework. Finally, an SMT-based scheduling approach is presented, followed by our conflict refinement approach. Note that multi-schedule synthesis is strongly constrained and backtracking is required. As heuristic approaches struggle with these late decisions we have selected an SMT-based approach that allows to efficiently solve the scheduling problem.

5.3.1 Methodology

The algorithm iteratively constructs a multi-schedule, starting with scheduling applications sharing processes for all variants. It then iterates over all subsets of variants from larger to smaller subsets, scheduling applications which have not been considered in previous iterations. This leads to a quickly increasing number of iterations with the number of variants. However, the complexity of the algorithm is dominated by the SMT-based schedule synthesis which is only applied for processes that have not been scheduled yet. While schedule synthesis might become intractable, efficient solvers such as Z3 [MB08] generally allow to solve moderate size problems in a reasonable amount of time. Hence, the reduction of the problem size for the scheduling algorithm through this iterative approach clearly out-weights any runtime increase. Our algorithm is based on the following parameters:

- $d \in D$ - variant specification describing a task graph $G_d = (P_d, E_d)$, representing the processes as vertices P_d and their data-dependencies as edges E_d. D denotes a set of all variant specifications. Described in detail in Definition 2.1.4.
- G_λ - comprehensive task graph $G_\lambda = (P_\lambda, E_\lambda)$, describing the relations of different variants to shared tasks, as illustrated in Figure 5.4(b) (page 121).
- $\beta(p) : P \rightarrow \mathbb{R}$ - multi-schedule defining the start-times of processes. It returns the process start-time $\mathbf{s}_p$ for a process $p \in P$, see Figure 5.4(c) (page 121).

Algorithm 5.1 outlines the outer-loop for our multi-schedule synthesis. The algorithm first initializes an empty comprehensive task graph $G_{\lambda'}$ and an empty schedule β (line 1-2), which are iteratively filled with processes that have been scheduled, and their assigned start-times, respectively. The algorithm iterates through all subsets $\tilde{D} \subseteq D$, starting from the complete set with $|D|$ elements down to each single variant (line 3-4). Processes which have already been scheduled are removed from $\tilde{D}$ (line 5), before a comprehensive task graph is generated (line 6). The comprehensive task graph G_λ abstracts the set of variants, containing all applications with shared processes. If a comprehensive task graph exists (line 7), a multi-schedule is generated, and β and $G_{\lambda'}$ are updated (lines 8-9). Figure 5.4 (page 121) illustrates two iterations with this algorithm.

The functions $\mathrm{getComprehensiveTaskGraph}(\tilde{D})$, determining a comprehensive task graph, and $\mathrm{generateMultiSchedule}(G_\lambda, G_{\lambda'}, \beta)$, determining a multi-schedule, are described

Algorithm 5.1: Outer-loop of multi-schedule synthesis

Input: set of variant specifications $d \in D$
Output: multi-schedule β

```
// initialize multi-schedule β, and comprehensive task graph for
   scheduled applications G_λ':
```
1 $\beta = \emptyset$
2 $G_{\lambda'} = (\emptyset, \emptyset)$
```
// iterate through all variant subsets D̃:
```
3 **for** $k \in \{|D|, .., 1\}$ **do**
4 **for each** $\tilde{D} \subseteq D$ **and** $|\tilde{D}| = k$ **do**
```
// only consider processes which have not been scheduled yet and
   calculate current comprehensive task graph G_λ:
```
5 $\tilde{D} = \{\tilde{d} := G_{\tilde{d}} = (P_d \setminus P_{\lambda'}, E_d \setminus E_{\lambda'}) | d \in \tilde{D}\}$
6 $G_\lambda(P_\lambda, E_\lambda) = \text{getComprehensiveTaskGraph}(\tilde{D})$
7 **if** $P_\lambda \neq \emptyset$ **then**
```
// determine multi-schedule β and update G_λ':
```
8 $\beta = \text{generateMultiSchedule}(G_\lambda, G_{\lambda'}, \beta)$
9 $G_{\lambda'} = (P_{\lambda'} \cup P_\lambda, E_{\lambda'} \cup E_\lambda)$
10 **end**
11 **end**
12 **end**

in detail in Sections 5.3.2 and 5.3.3-5.3.5, respectively. Note that if none of the variants shares any commonality with another variant, an individual schedule is created for each variant.

5.3.2 Determine Comprehensive Task Graph

This section proposes an algorithm to determine a comprehensive task graph. We define a comprehensive task graph as a task graph containing all vertices and edges of applications sharing a subgraph. Our algorithm uses the following additional parameter:

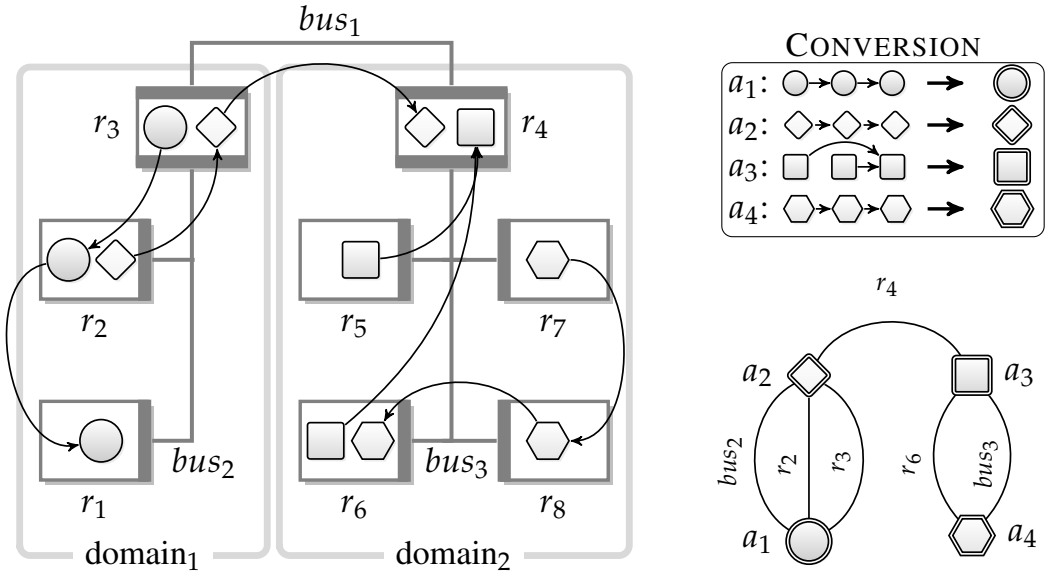
- $a \in A_d$ - defines an application with its task graph $G_a = (P_a, E_a)$. An application represents a weakly connected component of the task graph G_d of specification d ($P_a \subseteq P_d$, $E_a \subseteq E_d$), hence all processes $p \in P_a$ are connected through a path in the task graph. Described in detail in Definition 2.1.2.

To determine common subgraphs, usually the maximum common subgraph-isomorphism problem has to be solved which is known to be NP-hard [Lev73]. However, as our system model defines tasks and messages using very specific properties including task and message ids, we apply a significantly more efficient approach using sets.

Algorithm 5.2: Determine comprehensive task graphs

```
1  Function getComprehensiveTaskGraph (D̃)
       Input: subset of variant specifications D̃ ⊆ D
       Output: comprehensive task graph G_λ
       // determine processes P̃ common to all variants in D̃:
2      P̃ = ⋂_{d̃∈D̃} P_d̃
3      if P̃ ≠ ∅ then
           // initialize comprehensive task graph G_λ:
4          G_λ = (∅, ∅)
           // interate through all applications a of each variant d̃:
5          for d̃ ∈ D̃ do
6              for a ∈ A_d̃ do
                   // if a contains a process common to all variants in D̃,
                      add a to G_λ:
7                  if P_a ∩ P̃ ≠ ∅ then
8                      G_λ = (P_λ ∪ P_a, E_λ ∪ E_a)
9                  end
10             end
11         end
12         return G_λ
13     else
14         return (∅, ∅)
15     end
16 end
```

Algorithm 5.2 determines the comprehensive task graph for a set of variants. The algorithm first determines a common subset $\tilde{P}$ of processes shared by all variants in the subset $\tilde{D}$ (line 2) (see Figure 5.4(a), page 121). If a common induced subgraph exists (line 3), we initialize an empty comprehensive task graph G_λ (line 4) which is iteratively extended to the comprehensive task graph. The algorithm iterates through all variants (line 5), and their applications (line 6), determining if the application contains processes of the common subset $\tilde{P}$ (line 7). If the application shares processes in $\tilde{P}$, it is added to the comprehensive task graph G_λ (line 8). Finally, the algorithm returns G_λ (line 12), or if no common subset exists, it returns an empty task graph (line 14). Figure 5.4(b) (page 121) shows the comprehensive task graph G_λ for the three variants presented earlier. As the comprehensive task graph might contain parts that are not common to all variants in the current subset, the algorithm also maintains a set specifying the variants each process is part of.



(a) Domain-based architecture

(b) Application relations

Figure 5.6: *(a) Automotive Architectures are organized in domains, i.e., partitions, which only share few resources. (b) To partition the scheduling problem we convert the task graphs into a graph-based representation where each application a represents a vertex, and for each shared resource an edge is introduced between two applications.*

5.3.3 Partitioning

Before applying our variant-aware schedule synthesis, we apply a graph partitioning heuristic. It allows to apply a divide-and-conquer approach, scheduling each partition individually before re-integrating the generated schedules using schedule integration. Figure 5.6(a) illustrates two domains $domain_1$ and $domain_2$ of an automotive E/E-architecture, consisting of applications which are mainly executed on different resources, but share ECU r_4. Hence, the problem might be partitioned following this domain-based architecture. As shown in Chapter 4, partitioning of the problem to reduce the search space has proven beneficial to clearly improve the scalability of solving time-triggered scheduling problems.

Section 4.3.2 has proposed a partitioning approach to determine suitable partitions for the conflict refinement during schedule integration. However, while the purpose there was to select suitable partitions based on the cluster schedules, here we aim at selecting suitable clusters prior to schedule synthesis. To determine suitable partitions for the comprehensive task graph, we first introduce a graph-based representation to indicate application relations with regard to shared resources. We convert the comprehensive task graph $G_\lambda = (P_\lambda, E_\lambda)$ to a representation $G_{Cl_\lambda} = (A_{Cl_\lambda}, E_{Cl_\lambda})$ with applications A_{Cl_λ} as vertices and their resource dependencies as edges. Figure 5.6(b) illustrates such a graph for the applications illustrated in Figure 5.6(a). For instance, application a_2 and a_3 execute one task each on ECU r_4 and are therefore connected

by one edge. Similarly, a_1 and a_2 share three resources, r_2, r_3 and bus_2, resulting in three edges. For this example, a partitioning might be done by removing the edge between a_2 and a_3, generating two subgraphs. Schedules are then determined for each of the partitions $\{a_1, a_2\}$ and $\{a_3, a_4\}$ separately, and the partition schedules are integrated in a second step.

The partitioning is defined as:

$$\Lambda = \text{partition}(\lambda)$$

$$\tilde{\lambda} \in \Lambda,\ G_{\tilde{\lambda}} = (P_{\tilde{\lambda}}, E_{\tilde{\lambda}}) :\ P_{\tilde{\lambda}} \subseteq P_\lambda, E_{\tilde{\lambda}} \subseteq E_\lambda$$

Additionally we introduce the following parameter.

- $cl \in Cl_\lambda$ - cluster representing a subset of applications of variant specification G_λ as Graph $G_{cl} = (A_{cl}, E_{cl})$.

Metrics. The partitioning metrics presented in the following are not only limited to domains, but also determine subclusters within a domain. Our partitioning algorithm applies two metrics, a cost function evaluating the number of cuts required per application in a partition, and a balancing of the number of applications between the partitions. Balancing has proven beneficial in reducing partitions with single applications and leads to a similar processing time for each partition due to equal numbers of applications. Consequently, we define the balancing metric as:

$$\mu_{\text{imbalance}}(Cl_\lambda) = \underbrace{\frac{1}{|Cl_\lambda|} \cdot \sum_{cl \in Cl_\lambda}}_{\substack{\text{average over} \\ \text{all clusters}}} \sqrt{\left(\underbrace{\frac{|A_{cl}|}{\frac{|A_{Cl_\lambda}|}{|Cl_\lambda|}}}_{\substack{\text{deviation of } |A_{cl}| \\ \text{from average node} \\ \text{number per cluster}}} - 1 \right)^2} \tag{5.1}$$

With this metric we want to determine how much the number of nodes per cluster deviates from a perfectly balanced partitioning where each cl would obtain the same number of applications $|A_{cl}|$. Consequently, we first calculate the deviation of the number of vertices in each cluster compared to the average. Note that we subtract 1 in order to obtain the result of 0 if no imbalance exists and take the absolute value in case the imbalance calculation would result in a negative value. Finally, we calculate the average of the imbalance per cluster for the whole partitioning.

A partition that contains a high number of interlacing edges $e = (a, \tilde{a})$ between the subgraphs cannot be re-integrated efficiently. Therefore, we want to evaluate the number of crossing edges between partitions. Hence, we first define the function maxcross determining the maximum number of crossing edges between the subgraph and its adjacent subgraphs.

$$\text{maxcross}(cl) = \underset{|e|}{\text{argmax}} \left\{ |e| \;\middle|\; a_i \in A_{cl} \wedge a_j \in A_{\tilde{cl}} \right\} \text{ for all } \tilde{cl} \in Cl_\lambda \setminus cl \tag{5.2}$$

Here, we identify the maximum number of edges where vertices of a cluster are connected to vertices that are not in this particular cluster. Such edges are considered as crossing edges. Based on these considerations, we can determine a metric representing the average number of maximum crossing edges for all partitions in the graph.

$$\mu_{\text{avgcross}}(Cl_\lambda) = \frac{1}{|Cl_\lambda|} \sum_{cl \in Cl_\lambda} \text{maxcross}(cl) \tag{5.3}$$

Partitioning heuristic. Now that we can evaluate the quality of a graph partition, the actual algorithm that performs the partitioning, controlled by the two quality metrics, has to be defined. The graph shall be split such that a minimal imbalance $\mu_{\text{imbalance}}(Cl_\lambda)$ is achieved while the subgraphs have an acceptable number of interlacing edges $\mu_{\text{avgcross}}(Cl_\lambda)$:

$$\text{minimize } \mu_{\text{imbalance}}(Cl_\lambda) \text{ s.t. } \mu_{\text{avgcross}}(Cl_\lambda) < \epsilon_{\text{cross}} \tag{5.4}$$

For this purpose, we apply the efficient polynomial time Girvan-Newman algorithm [GN02] which determines subgraphs as communities with a higher number of connections between the vertices by iteratively removing edges from the graph until subgraphs are formed. Here, the number of removed edges per vertex is the parameter influencing if the initial graph is split at all, and how many subgraphs are created.

We start with the edge-removal parameter set to 1 and increase the number of removed edges per vertex by 1 in each iteration. For each iteration we calculate $\mu_{\text{imbalance}}(Cl_\lambda)$ and $\mu_{\text{avgcross}}(Cl_\lambda)$ and decide whether the result of the graph partitioning is suitable according to our metrics. Consequently, we only start to evaluate $\mu_{\text{imbalance}}(Cl_\lambda)$ once the first splitting in the graph has occurred, as an unpartitioned graph is inherently balanced. We only accept a partition if $\mu_{\text{avgcross}}(Cl_\lambda)$ is below a threshold of ϵ_{cross}. As $\mu_{\text{avgcross}}(Cl_\lambda)$ is either the same or increases between two iterations, this metric can result in suggesting that the graph should be processed as a whole.

As we monitor $\mu_{\text{imbalance}}(Cl_\lambda)$ while iterating the number of removed edges, we compare its value to the result from the previous step. The algorithm is designed to achieve a certain balance after some iterations, before $\mu_{\text{imbalance}}(Cl_\lambda)$ again increases. Hence, we use the last splitting result before the imbalance in the graph increases.

Preprocessing. So far the proposed partitioning heuristic assumes that domains are also mapped to separate hardware domains, and hence, that most applications in different domains do not share resources. However, if multiple domains share the same communication bus, this is not given anymore and all applications share a common edge, forming a clique. To partition such graphs, we apply a preprocessing and remove the edges of a shared communication bus before applying our partitioning heuristic.

Schedule integration. After partitioning the comprehensive task graph to subgraphs, we apply

the schedule synthesis described in Section 5.3.4 for each subgraph individually. To re-integrate the individual partition schedules into a global schedule, we apply the schedule integration approach described in Chapter 4.

5.3.4 Schedule Synthesis

In the following, we present an SMT-based multi-schedule synthesis. Applied to the generated comprehensive task graph, it schedules applications of multiple variant specifications in parallel, generating a multi-schedule as depicted in Figure 5.4(f) (page 121). The multi-schedule represents all individual variant schedules.

We assume a bus that allows to freely allocate time slots, such as for Automotive Ethernet. As the focus here is on variant-aware scheduling, we do not explicitly consider a switched network supporting a concurrent message transmission. However, an extension of the proposed approach to support a switched Ethernet network is possible (see Sections 3.2.4 and 4.3.3). The schedule synthesis is based on the following additional parameters:

- p - process, referring to both tasks and messages. Defined in detail in Definition 2.1.1.
- h_p - process period, time after which p is repeated.
- τ_p - execution-time for a task or transmission time for a message.
- $r(p) : \ P \rightarrow R$ - returns the predefined process mapping to a resource. The set of resources R consists of both ECUs and communication buses. Defined in detail in Definition 2.1.3.
- $D(p) : \ P \rightarrow \mathcal{D}$ - returns all variant specifications containing a process p.
- $h(p,\tilde{p}) = \text{lcm}(h_p, h_{\tilde{p}}) : P \rightarrow \mathbb{R}$ - returns hyper period of p and $\tilde{p}$ where $\text{lcm}(\cdot)$ defines the least common multiple. Defines period after which the schedule for p and $\tilde{p}$ repeats.
- $e = (p,\tilde{p}) \in E$ - data-dependency of $\tilde{p}$ from p, defining the process precedence.
- $\theta_a \in \mathbb{R}$ - deadline of application a. Defines maximum end-to-end delay from all source to all sink processes.
- ϕ - path or subgraph from a source to a sink task, e.g., $\phi = \{p_1, p_2, .., p_n\}$. The processes must be pairwise connected with an edge e. Defined in detail in Definition 2.2.4.
- $\Phi(E_a) : \mathcal{E} \rightarrow \{\Phi\}$ - returns all paths ϕ of application a. Applies a *Depth-first search* algorithm to determine paths [Eve11].

Based on these parameters our schedule synthesis determines a schedule using the following variables.

- $\mathbf{s}_p \in \mathbb{R}$ - variable for start-time of p.
- $\mathbf{f}_p \in \mathbb{R}$ - variable for finish-time of p.
- $\mathbf{w}_{(p,\tilde{p})} \in \mathbb{R}$ - variable for waiting-time between two data-dependent processes $p, \tilde{p}$, defined as the delay between the finish-time of p and start-time of $\tilde{p}$.
- $\mathbf{o}_a \in \mathbb{R}$ - variable for application offset, applied to applications scheduled in a previous iteration.

The following constraints determine a schedule for the comprehensive task graph G_λ.
$\forall p \in P_\lambda$:

$$0 \leq \mathbf{s}_p < h_p \tag{5.5}$$

$\forall p, \tilde{p} \in P_\lambda, p \neq \tilde{p}, D(p) \cap D(\tilde{p}) \neq \emptyset, r(p) = r(\tilde{p}),$
$i = \{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_p} - 1\}, j = \{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\}$:

$$\begin{aligned} & i \cdot h_p + \mathbf{s}_p + \tau_p \leq j \cdot h_{\tilde{p}} + \mathbf{s}_{\tilde{p}} \\ \oplus \quad & j \cdot h_{\tilde{p}} + \mathbf{s}_{\tilde{p}} + \tau_{\tilde{p}} \leq i \cdot h_p + \mathbf{s}_p \end{aligned} \tag{5.6}$$

$\forall a \in A_\lambda, e \in E_a, (p, \tilde{p}) \in e$:

$$\mathbf{f}_p = (\mathbf{s}_p + \tau_p) \bmod h_p \tag{5.7}$$

$$\mathbf{w}_{(p,\tilde{p})} = (\mathbf{s}_{\tilde{p}} - \mathbf{f}_p) \bmod h_p \tag{5.8}$$

$\forall a \in A_\lambda, \phi \in \Phi(E_a)$:

$$\theta_a \geq \sum_{p \in \phi} \tau_p + \sum_{(p,\tilde{p}) \in \phi} \mathbf{w}_{(p,\tilde{p})} \tag{5.9}$$

Here, the symbol $\oplus$ represents an *exclusive or* (XOR) operation. Constraint (5.5) defines the boundaries for the start-time. As processes are periodically executed, their start-time cannot exceed their process period h_p (cf. Definition 2.2.5). Constraint (5.6) ensures that two processes in the same variant specification are not executed at the same point in time if both are mapped to the same resource, and therefore finish execution before another process is started for any of their periods (cf. Requirement 2.2.1). Constraint (5.7) calculates the finish-time $\mathbf{f}_p$. Constraint (5.8) determines the waiting-time between two data-dependent processes $p, \tilde{p}$. Finally, Constraint (5.9) ensures that the maximum delay based on the execution and waiting-times of an application for each respective path does not exceed the application deadline (cf. Require-

ment 2.2.3). The modulo operation mod in Constraints (5.7) and (5.8) is defined as a function in our SMT formulation with the following properties:

$$0 \leq i \bmod j < j$$

Hence, for $i < 0$ it returns a positive value, e.g., for $-j < i < 0 : i + j$. This ensures that the values for both the finish- and the waiting-time are positive, satisfying Requirement 2.2.2

Schedules of previous iterations. Our multi-schedule approach iteratively generates an individual schedule for each variant, scheduling common parts first, before extending each variant schedule with variant specific parts in a consecutive iteration. To take parts of the schedule created in a previous iteration into account, additional constraints are required. The processes $P_{\lambda'}$ scheduled in previous iterations are considered as temporal intervals where processes of the current comprehensive task graph P_λ cannot be scheduled if they are part of the same variant and are mapped to the same resource, see grayed out parts in Figure 5.4(f) (page 121). To allow modifications to applications scheduled in a previous iteration, we introduce an application offset $\mathbf{o}_a$, defining a common offset for all processes $p \in P_a$. As all task and message start-times are adjusted concurrently through this offset, the general structure of the application schedule is not altered and therefore all previously defined constraints are not affected. Corresponding to Constraint (5.6), we define the following constraints:

$\forall a \in A_{\lambda'}$:

$$0 \leq \mathbf{o}_a < h_a \tag{5.10}$$

$\forall p \in P_\lambda, \forall a \in A_{\lambda'}, \tilde{p} \in P_a, D(p) \cap D(\tilde{p}) \neq \emptyset, r(p) = r(\tilde{p}),$
$i = \{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_p} - 1\}, j = \{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\} :$

$$\begin{aligned} & i \cdot h_p + \mathbf{s}_p + \tau_p \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o}_a \\ \oplus \quad & j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o}_a + \tau_{\tilde{p}} \leq i \cdot h_p + \mathbf{s}_p \end{aligned} \tag{5.11}$$

$\forall a, \tilde{a} \in A_{\lambda'}, p \in P_a, \tilde{p} \in P_{\tilde{a}}, D(p) \cap D(\tilde{p}) \neq \emptyset, r(p) = r(\tilde{p}),$
$i = \{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_p} - 1\}, j = \{0, .., \frac{2 \cdot h(p,\tilde{p})}{h_{\tilde{p}}} - 1\} :$

$$\begin{aligned} & i \cdot h_p + s_p + \mathbf{o}_a + \tau_p \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o}_{\tilde{a}} \\ \oplus \quad & j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o}_{\tilde{a}} + \tau_{\tilde{p}} \leq i \cdot h_p + s_p + \mathbf{o}_a \end{aligned} \tag{5.12}$$

Constraint (5.10) defines the boundaries for the application offset. Constraint (5.11) ensures that a currently scheduled process does not intersect with already scheduled processes. The application offsets $\mathbf{o}_a$ hereby allow to alter the structure of parts which have already been

scheduled in a previous iteration. Finally, Constraint (5.12) ensures that altering the application offsets does not lead to intersections between previously scheduled applications.

5.3.5 Conflict Refinement

Iterative schedule synthesis might lead to conflicts which manifest in the currently scheduled applications not being combinable with a schedule generated in a previous iteration. We therefore propose a conflict refinement which determines conflicting parts and resolves the conflict through adapting the schedules. Our conflict refinement approach first determines conflicting application schedules and updates the comprehensive task graph G_λ and the comprehensive task graph of previously scheduled processes $G_{\lambda'}$ to resolve the conflicts, as illustrated in the design flow in Figure 5.5 (page 123).

To determine conflicting application schedules in G_λ and $G_{\lambda'}$, we determine all IISs using the extended deletion filter presented in Section 4.3.2.1. An IIS represents a smallest set of conflicting constraints which might be resolved by removing any of these conflicting constraints. The extended deletion filter removes groups of constraints until the remaining set of constraints is feasible. For multi-schedule synthesis, the constraint groups are defined by applications. Thus, the extended deletion filter iteratively removes the constraints of one application from the schedule synthesis problem until it is solvable. The last removed application is then returned to the set and the process is continued until the remaining problem is unfeasible and removing any of the conflicting applications would resolve the conflict. However, as the schedule synthesis for G_λ might contain multiple IISs, we apply the deletion filter multiple times until all IISs have been determined.

To resolve the conflicts determined with the IISs, we extend G_λ with the conflicting applications, $G_\lambda = (P_\lambda \cup P_{\text{IISs}}, E_\lambda \cup E_{\text{IISs}})$, and remove these applications from $G_{\lambda'}$ for all IISs, $G_{\lambda'} = (P_{\lambda'} \setminus P_{\text{IISs}}, E_{\lambda'} \setminus E_{\text{IISs}})$. The schedule synthesis is then applied to the updated G_λ and $G_{\lambda'}$.

5.4 Experimental Results

To evaluate our proposed multi-schedule framework, we first compare the resource requirements of a global schedule, i.e., one single schedule for all variants, to our multi-schedule approach. Second, we analyze the deviation of variant schedules created by a variant-unaware ILP approach, i.e., generating an individual schedule for each variant ignoring commonality, compared to our framework. Third, a scalability analysis evaluates our framework in comparison to a variant-aware ILP. Finally, we present a case study using an experimental prototype of an automotive architecture which allows to adapt the architecture and switch the system schedule. The schedule synthesis has been carried out on an Intel Xeon 3.2 GHz Quad Core with 12GB RAM. We use Microsoft's Z3 version 4.3.0 as SMT solver for multi-schedule synthesis [MB08]. For the graph partitioning, we have determined a threshold of $\epsilon_{\text{cross}} = 1.3$ for our

average crossing metric (see Section 5.3.3) as beneficial through an experimental analysis. For the ILP approach we apply the ILP formulation defined in Section 3.2. Note that the schedule is obtained at design time such that runtimes of several minutes are still acceptable.

5.4.1 Resource Utilization

To generate time-triggered schedules for different variants, car manufacturers commonly generate a single global schedule, including the tasks and messages of all variants, to reduce testing and integration efforts. This leads to the allocation of resources for tasks which are not executed in each variant. Therefore, the number of applications which might be deployed in a variant is limited through the worst case resource utilization assumption of the global schedule, while the actual resource utilization is low. For instance, if we consider a system with two variants, sharing one common application, such as ABS, and having variant specific applications, like the motor control for a diesel and a petrol engine, then the global schedule needs to accommodate three applications in a single schedule, i.e., the ABS application and both the diesel as well as the petrol application. By contrast, the multi-schedule distributes these three applications as two applications in each variant schedule. In the following, we analyze the number of applications supported by a set of variants if scheduled by a global schedule and our framework. To generate a global schedule, we apply the non-variant-aware ILP approach from Section 3.2 which adds both common as well as individual tasks and messages to a single schedule. We use CPLEX in version 12.6 as ILP solver [ILO].

We have created a set of synthetic applications of which 60 % are common, hence, these applications are added to all variant schedules. The remaining applications are variant specific and are only part of a single variant[2]. For a given small subsystem, we randomly select applications from this set and iteratively add them to the subsystem, until no feasible solution can be found within a timeout of 5 minutes. For the global schedule, both common and individual applications are added to the scheduling problem. For our multi-schedule synthesis only common applications are added to all variant schedules while individual applications are only added to one variant schedule. The metric applied for this analysis is the number of applications supported by the schedule synthesis approaches.

Figure 5.7 shows the average results for 3200 synthetic test cases. The results clearly show the benefits of multi-schedules compared to a global schedule. The global schedule only supports a limited number of approx. 3.5 applications on average which is independent of the number of variants. By contrast, a multi-schedule uses the available resources significantly more efficiently, and the number of supported applications increases with the number of variants. Note that common and individual applications are selected randomly such that for this case study the resulting average application number supported by a multi-schedule might be lower

[2]An application consists of 3 to 7 tasks communicating via up to 6 messages. For instance, application a_1 in specification d_1 in Figure 5.2 consist of 3 tasks communicating via 2 messages, having one sensor and one actuator task. For this case study we also consider applications with multiple sensor and actuator tasks sending and receiving multiple messages. All task periods are harmonic and range from 5 ms to 80 ms.

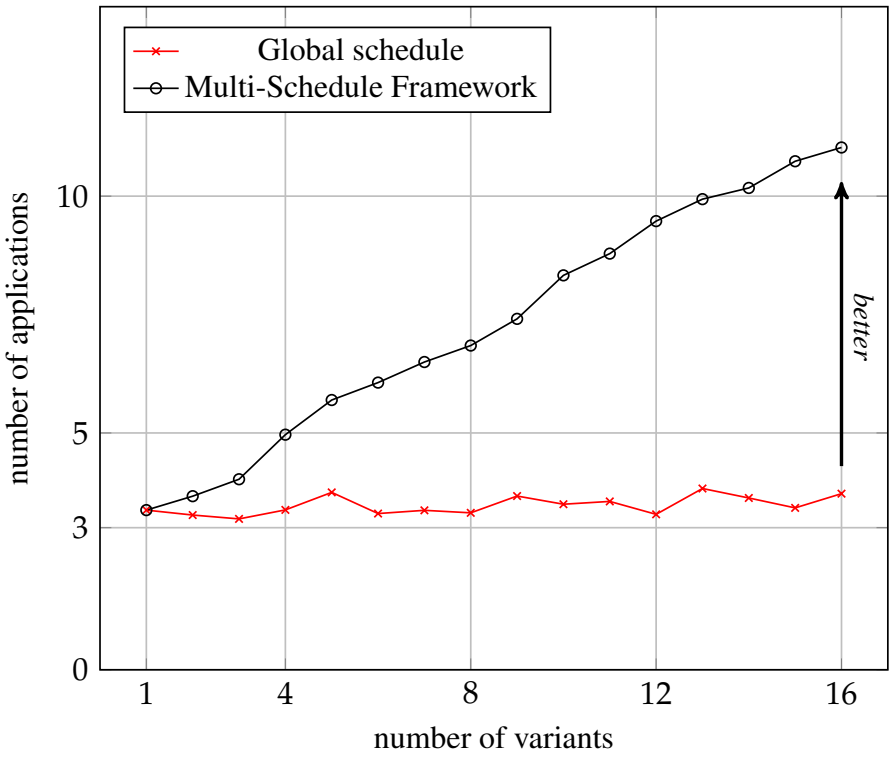


Figure 5.7: *Comparison of application number supported by a single global schedule for all variants and a multi-schedule generating an individual schedule for each variant. A higher number of applications for a set of variants indicates a more effective resource utilization. The average values obtained for* 3200 *synthetic test cases are shown.*

than the number of variants. These results indicate to which extent the worst case resource utilization assumption of the global schedule overestimates the actual resource requirements, leading to a poor resource utilization. By contrast, the generation of multi-schedules leads to a clearly improved resource utilization and in consequence allows to deploy the same applications on an architecture with less ECUs.

5.4.2 Analysis of Variant-Awareness

To ensure an efficient resource utilization, for each variant a schedule could be created independently. However, as each variant schedule is created individually, the schedules strongly differ. This leads to significantly increased testing and integration efforts as common applications have to be tested for each variant individually. To evaluate the differences between independently created variant schedules, in the following, we compare the results of a non-variant-aware ILP with our framework. The single-stage ILP is applied to each variant independently. Hence, in contrast to our framework, the resulting variant schedules do not have the same start-times assigned to shared tasks.

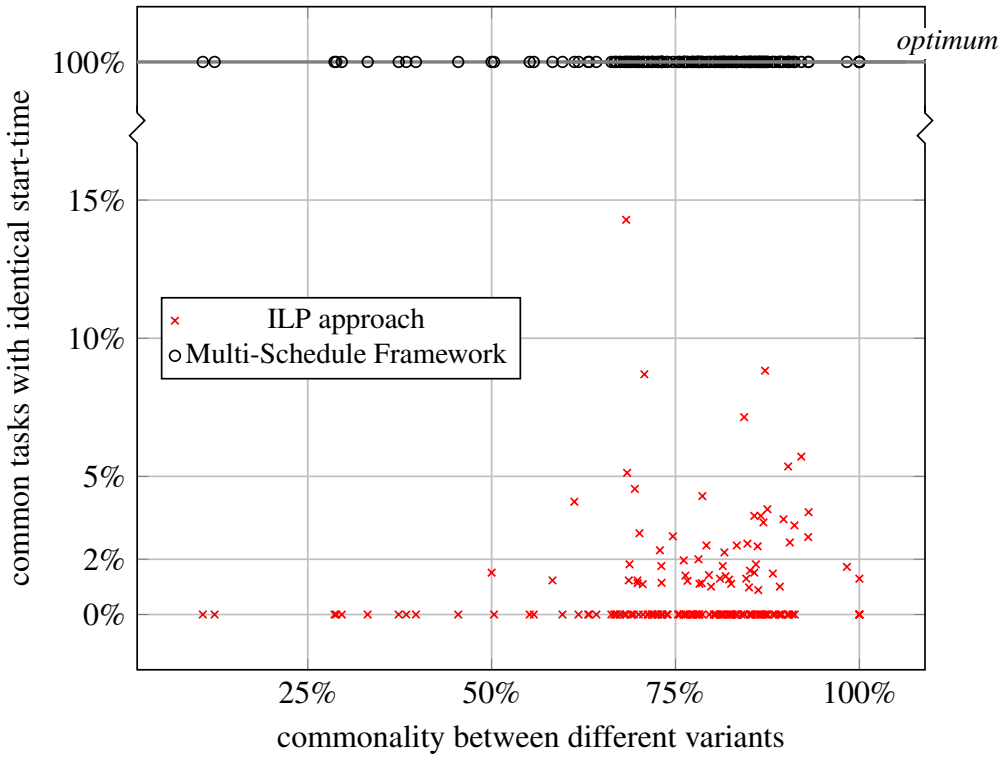


Figure 5.8: *Analysis of differences in schedules created by a non-variant-aware ILP approach and our variant-aware multi-schedule framework, depending on the ratio of common tasks to all tasks including variant specific tasks. We consider tasks to share the same schedule if their start-times differ less than* 0.1 ms.

To evaluate the approaches, we use 210 synthetic test cases with hardware architectures consisting of up to 20 ECUs connected by an Ethernet bus. A test case consists of 40 to 446 tasks and messages which are distributed in 4 to 16 variants[3]. The number of processes common to the variants range from 10 % to 100 % of all tasks and messages. In the following, we first analyze the differences in variant schedules determined by the non-variant-aware ILP compared to our framework, before presenting a runtime and an end-to-end delay analysis of both approaches.

Variant-awareness. Figure 5.8 illustrates the number of shared tasks with identical schedules for different variants. The results for the ILP show that for more than 65 % of the test-cases the start-times for all tasks differ more than 0.1 ms. A detailed analysis has shown that the average difference for common parts lies between 2 ms and 33 ms for the ILP, indicating a clear difference between common parts of variant schedules. With an increasing commonality between different variants, the number of identical schedules for common tasks might increase slightly, but stays below 10 % for most cases. These results do not come as a surprise, as

[3] A single variant might consist of up to 131 tasks and messages. All tasks have harmonic periods ranging from 5 ms to 80 ms.

each variant schedule is generated individually and common start-times are by chance and not intentional. Our framework instead assigns an identical schedule to all shared tasks. These results show the necessity of a variant-aware approach to reduce testing and integration efforts.

Runtime evaluation. Evaluating the runtimes for this case study shows that for 80 % of the test cases our framework is faster in creating a multi-schedule than the non-variant-aware ILP in generating all variant schedules individually. On average, the ILP calculation takes 7 times longer than the multi-schedule synthesis to determine a schedule for each variant. This result shows the efficiency of our framework to deal with the increased complexity of variant-aware scheduling in comparison to conventional non-variant-aware approaches. Note that this runtime only accounts for the time required for the schedule synthesis but does not quantify the time savings achievable through reduced testing and integration efforts.

End-to-end delay analysis. To evaluate the drawbacks of multi-schedule synthesis, we compare the overall end-to-end delay of the applications in all variants for 120 of the test cases for which both approaches find a solution. Both approaches ensure that the maximum end-to-end delays defined for each application are satisfied. However, generating an individual schedule for each variant leads to a lower end-to-end delay for 72 % of the test cases. On average, the determined multi-schedules have an end-to-end delay which equals 94.0 % of the maximum end-to-end delays, while generating each variant schedule individually leds to 85.5 %. These results indicate that the additional constraints imposed by the concurrent schedule synthesis might lead to an increased application end-to-end delay and consequently to a reduced control function performance. However, if this reduction in system performance is acceptable, variant-aware schedule synthesis helps to reduce development costs as testing and integration efforts are reduced. This is generally the case for automotive applications which are robust and perform well with an increased end-to-end delay, as long as their maximum end-to-end delay is not exceeded. These results are obtained without applying an objective function.

5.4.3 Runtime Analysis

The previous two case studies have evaluated the benefits of the proposed multi-schedule synthesis compared to a single global schedule and a non-variant-aware approach. As these approaches are not completely comparable with multi-schedule synthesis, in the following, we compare our framework with a variant-aware ILP approach to evaluate the efficiency. The ILP approach generates all variant schedules in a single iteration, taking commonality into account. We evaluate the approaches using the same 210 synthetic test cases as introduced in the previous section.

Figure 5.9 shows the results of the runtime analysis. We have defined a timeout of 15 minutes and consider test cases which cannot be solved within this time frame as infeasible. The results show that our multi-schedule framework performs well compared to the variant-aware

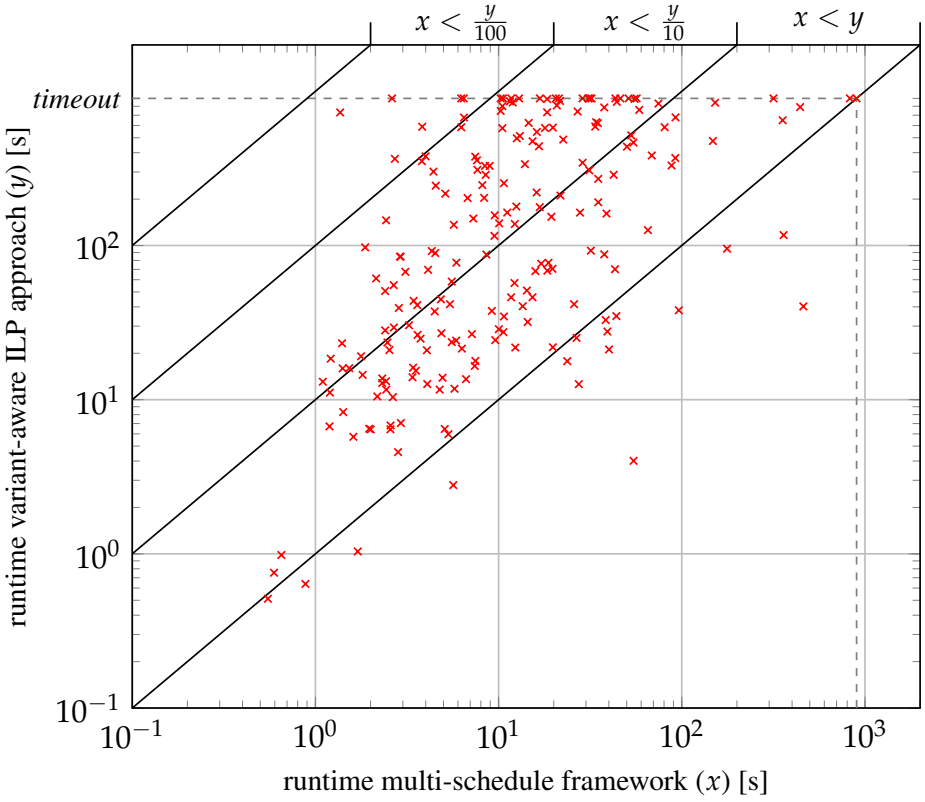


Figure 5.9: *Runtime comparison of our framework generating a multi-schedule with a variant-aware ILP approach generating all variant schedules concurrently.*

ILP. While for some test cases our framework might introduce an overhead, in particular for difficult test cases it outperforms the ILP. On average, our framework is 14 times faster, showing the benefits of our iterative approach in combination with the problem partitioning. For various test cases the ILP is unable to find a solution within 15 minutes while our framework finds a solution. For the multi-schedule synthesis, the framework requires 4 iterations on average to determine all variant schedules for one test case, but not more than 11. About 74 % of all test cases require a conflict refinement. The problem partitioning proposed in Section 5.3.3 accounts for a runtime reduction of 21.4 % for all test cases and 72.6 % for test cases with a runtime of over 120 seconds without partitioning. This indicates that, in particular for large and difficult test cases, the partitioning clearly improves the scalability of our approach.

5.4.4 Automotive Case Study

To illustrate the importance of a variant-aware schedule synthesis, in the following, we apply multi-schedule synthesis for the lab setup of a time-triggered automotive architecture. The system is able to adapt the hardware architecture and switch between predefined schedules at

runtime. This allows to evaluate multiple variant schedules on the platform. To minimize the changes of the system configuration induced when switching schedules, shared tasks require an identical schedule for different variants. As multi-schedule synthesis fulfills this property, this experimental prototype provides an ideal testbed to evaluate our framework.

In the area of *multi-mode* systems various work has been done on switching between system configurations. In this context, we address the problem of minimizing the differences between modes, i.e., variant schedules. However, the objective of this case study is not to obtain minimal switching delays, but rather to evaluate multi-schedule synthesis. A more detailed discussion of related work was given in Section 5.1.

We have defined 9 variants for our experimental prototype, of which 3 are shown in Figure 5.2 (page 115). The lab setup allows to turn off single ECUs, thus, changing the underlying architecture. Each ECU runs an online diagnosis detecting deactivated ECUs. We use this mechanism to switch the current system schedule and to evaluate our framework. In the following, we present the results obtained with our multi-schedule synthesis and the non-variant-aware ILP from Section 3.2. Note that this case study gives an example where determining a single global schedule is not feasible, as a global schedule is unable to accommodate the applications of all 9 variants.

Scheduling results. The 9 variant schedules have been generated with both our framework and the variant-unaware ILP introduced in the previous section. To generate all variant schedules, our framework takes 50 ms while the ILP takes 90 ms. Here, the incremental approach and scheduling multiple variants concurrently reduce the runtime compared to the ILP approach. As we have predefined the start-times of the system tasks *PTP* and *FD*, the schedule synthesis was applied only for the applications *STR* and *IMG*. While the variant schedules generated with our framework (Figure 5.10) take the commonality of multiple variants into account, the schedules generated by the ILP (Figure 5.11), clearly differ in the start-times for common parts. For instance, while our framework has assigned the same start-times for the shared tasks t_{STRPROC} and m_1, their start-times differ for all schedules generated by the ILP, e.g., t_{STRPROC} is scheduled with $s_{\text{STRPROC}} = 1.2\,\text{ms}$, $s_{\text{STRPROC}} = 3.0\,\text{ms}$, and $s_{\text{STRPROC}} = 0.6\,\text{ms}$, in Figure 5.11(1),(2) and (3), respectively. Figure 5.11 shows the differences between schedules generated individually, requiring individual testing and integration of all variant schedules. By contrast, our framework generates homogeneous variant schedules, as illustrated in Figure 5.10, clearly reducing the testing and integration efforts as schedules for common applications only need to be tested once, and only distinctive applications require individual testing.

Trade-offs. These results also indicate limitations of our approach. For instance, while we have not defined a minimal end-to-end delay as an objective for the schedule synthesis, for the variant schedules illustrated in Figure 5.11, generating each schedule individually has led to a reduced end-to-end delay for the application *STR* compared to the results obtained by a multi-schedule in Figure 5.10. This can in particular be seen in Figure 5.10(3) where a delay is introduced

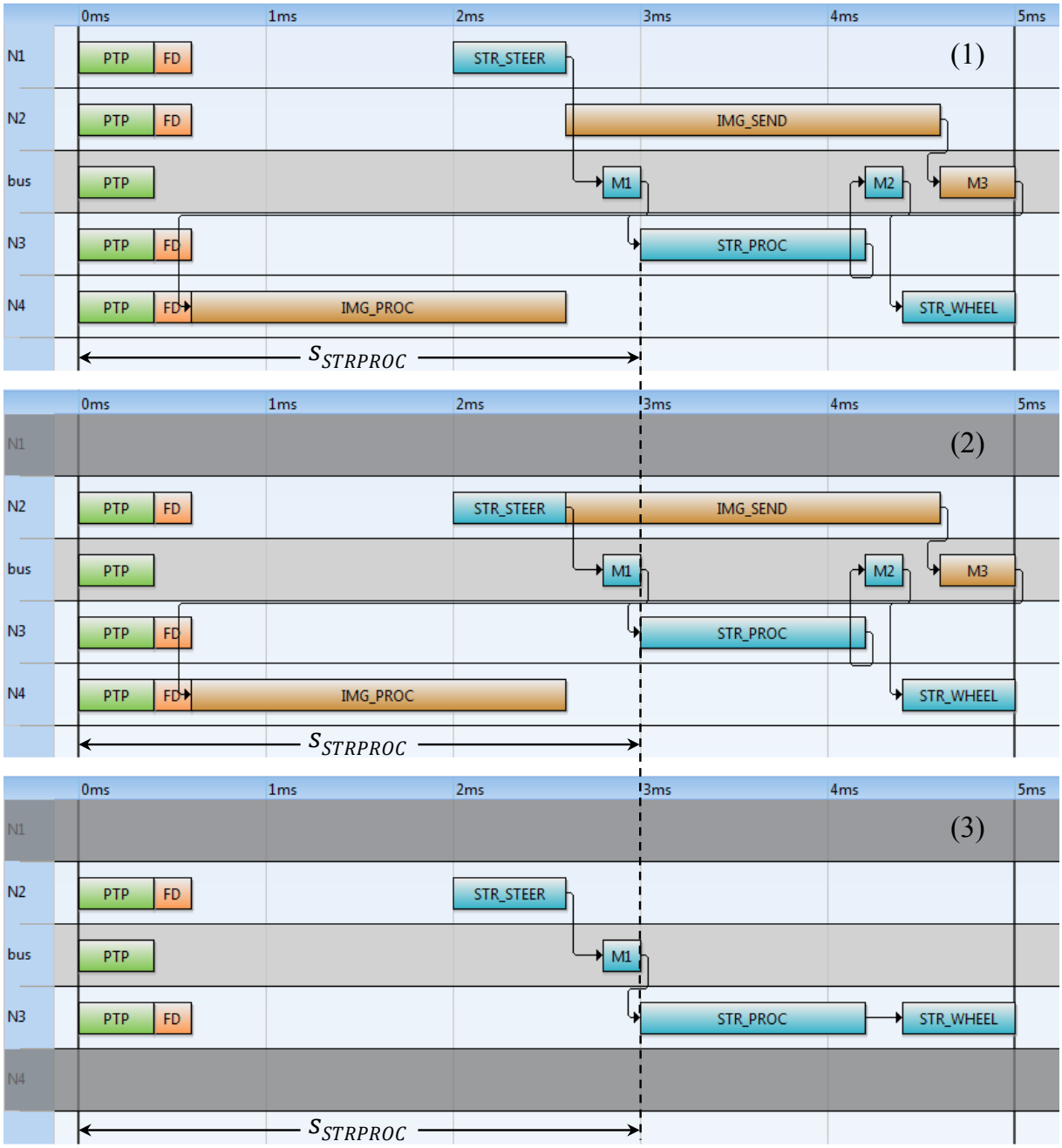


Figure 5.10: *Schedules for three variants created by our framework. The variant task graphs are defined in Figure 5.2. Note that we follow a periodic execution model. Hence, sink processes (e.g.* t_{IMGPROC}*) might appear to be executed before source processes (e.g.* t_{IMGSEND}*), however, in this case data generated in the previous period is processed.*

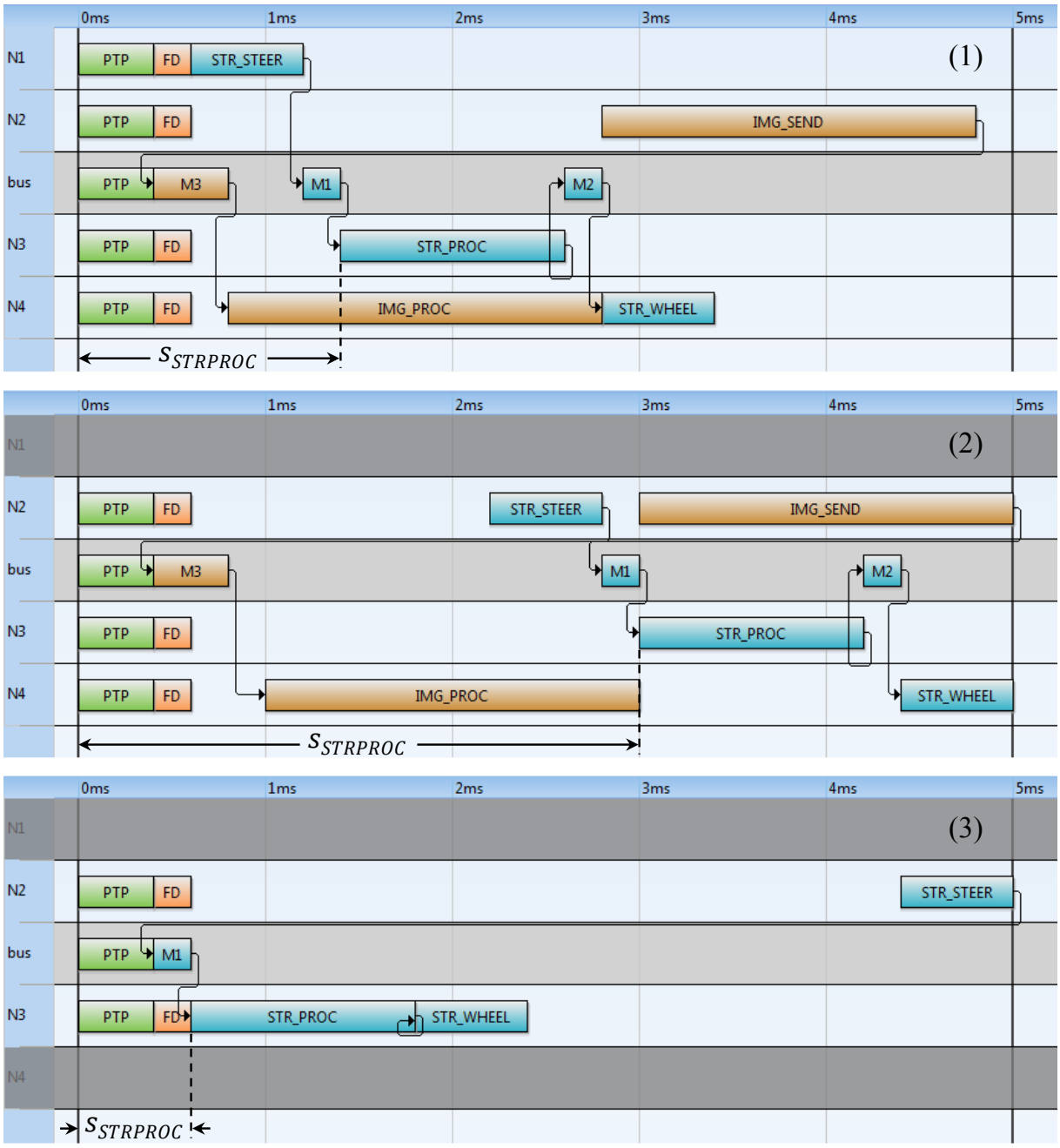


Figure 5.11: *Variant schedules determined by an ILP approach, generating variant-unaware schedules.*

between t_{STRPROC} and t_{STRWHEEL} for m_1 scheduled in the other variants. Hence, variant-aware scheduling might lead to an increased end-to-end delay, and consequently to a reduced control function performance. However, as our approach ensures that all defined maximum end-to-end delays are not violated, the control performance might be slightly reduced but the correct functionality is guaranteed while the testing and integration efforts are significantly reduced.

5.5 Summary

This chapter addressed the problem of generating multi-schedules for variant management in time-triggered architectures. A multi-schedule defines an individual schedule for each variant which shares the same schedule for common parts of different variants. We proposed a framework for multi-schedule synthesis which determines commonality in multiple variants and calculates an identical schedule for these common parts. We apply an incremental approach which also considers commonality in variant subsets. To improve the scalability of our approach, we also presented a partitioning heuristic, generating subproblems which might be solved independently and are re-integrated using schedule integration.

The experimental results, consisting of an extended analysis of resource requirements, the deviation between variants, and scalability as well as an automotive lab setup, showed the benefits of our approach. As our approach generates variant-specific schedules, the resource requirements are clearly improved as compared to a single global schedule, used in all variants. Multi-schedule synthesis only allocates the resources which are required in each variant. At the same time, the deviation between variants is significantly reduced compared to a variant-unaware approach, generating individual schedules without taking commonality into account. This leads to a clear reduction of testing and integration efforts compared to a variant-unaware approach, as applications can be designed independently of the variant they are deployed in. These results show the need for an efficient variant-aware approach.

The graph-based partitioning heuristic presented in this chapter gives an example on how suitable clusters might be selected for the schedule integration presented in Chapter 4. It allows to identify subsets of applications which have minimal correlations, both aiding the development as well as the integration process.

In this chapter we have shown the benefits of a variant-aware schedule synthesis and proposed an efficient implementation based on multi-schedule synthesis. The focus here was on minimizing the differences between variants. Taking into account additional objectives such as control performance might therefore be an interesting extension of our work. In addition, here the focus was on time-triggered systems. An important line of future work is therefore a variant-aware approach considering event-triggered tasks and messages.

6 Concluding Remarks

With an increasing number of advanced driver assistance systems and the upcoming drive-by-wire, a clear trend towards time-triggered architectures can be seen in the automotive domain. This thesis investigated various aspects of schedule synthesis for time-triggered systems, with a focus on automotive E/E-architectures.

6.1 Summary

This thesis first addressed schedule synthesis for message transmission in the static segment of the automotive FlexRay bus. The focus was on developing new methods that support all new features of the latest FlexRay version 3.0, while being applicable to the previous version 2.1 to the same extent. We presented various approaches: An ILP generating a schedule with a minimal bandwidth utilization, a heuristic approach with a good scalability and an ILP-based heuristic approach which integrates individually generated FlexRay schedules in a common schedule. An extensive case-study showed the benefits of the proposed approaches compared to GA and SA approaches. The results also indicate the potential of FlexRay 3.0 to make the bandwidth utilization more efficient. Furthermore, we also showed how this asynchronous FlexRay scheduling, focusing on message transmission, can be extended to a holistic scheduling that takes both tasks and messages into account.

Second, a modular schedule synthesis for task and message scheduling was proposed. The schedule integration framework allows to integrate previously generated subsystem schedules into a global schedule. To improve the scalability and minimize the changes required to subsystem schedules, a multi-stage approach was presented. The framework first tries to integrate

all subsystem schedules without altering their structure. As integrating individually generated subsystem schedules in a single schedule might not always be feasible, we also proposed a conflict refinement which identifies conflicting schedules and resolves the conflict by altering the subsystem schedules. A partitioning heuristic reduces the complexity of the conflict refinement process through identifying subproblems which might be solved individually. The framework supports both Automotive Ethernet as well as FlexRay. To facilitate the schedule integration, we also investigated several integrability metrics which optimize subsystem schedules for schedule integration. The results showed the significantly better scalability of schedule integration compared to a conventional holistic scheduling. An extended discussion illustrated how schedule integration might enable a highly composable system, allowing to easily update and add applications. We also discussed how this design approach might be integrated with an AUTOSAR design flow.

Finally, a multi-schedule synthesis approach was presented which determines schedules for different system variants. The proposed framework generates an individual schedule for each variant, while taking into account commonality between variants. Thus, the individual variant schedules share an identical schedule for common parts. The framework not only identifies tasks and messages common to all variants, but also determines commonality between subsets of variants. We apply an iterative approach to handle the increased complexity of a variant-aware schedule synthesis. In addition, a partitioning heuristic was proposed which allows to determine suitable clusters for which a schedule might be determined individually. Schedule integration then allows to integrate the cluster schedules in a common multi-schedule. The results give evidence of the benefits of the proposed approach, both in regards to variant-awareness as well as scalability.

6.2 Future Work

The proposed schedule synthesis approaches are extensible to a large extent. In the following, potential future work is discussed.

Asynchronous FlexRay scheduling. The FlexRay scheduling framework proposed in Section 3.1 addresses the problem of message scheduling of a single bus. However, in the automotive industry it is common that applications communicate with multiple domains. Thus, a message might not only be sent on a single bus but is forwarded to other buses via gateways. A future extension might therefore consider multiple buses. An additional objective would then be to pack messages to slots such that message repacking in the gateways is minimized.

Schedule integration. Several extension possibilities exist for the schedule integration framework presented in Chapter 4. While the framework allows to efficiently integrate cluster schedules without altering the general structure, the conflict refinement suffers from a limited scalability if partitioning the problem is not feasible. Improving the scalability of the conflict refine-

ment would therefore be a valuable extension to the schedule integration framework. For the SMT-based approach, selecting suitable processes for which the offsets are adjusted instead of considering all processes might already lead to a clear runtime reduction, as the search space is reduced. However, also investigating if heuristic approaches are suitable for the conflict refinement might be an interesting direction for future work.

Moreover, we have shown several metrics to optimize the cluster schedules for their integrability. Combining several of these metrics to define an objective allows to determine competitive results. An extensive analysis might allow to determine suitable weights for each metric and further improve the integrability of cluster schedules.

Incorporating knowledge of other clusters during cluster schedule generation would also allow to further improve the integrability of clusters. Developing suitable metrics, accounting for either the structure of a legacy cluster schedule or information of task and message distributions in other clusters, is therefore an interesting line of future work.

Furthermore, the schedule integration framework currently only considers time-triggered systems. However, automotive architectures implement a mix of event-triggered and time-triggered systems depending on the application. Thus, a valuable extension of our framework would be the support of mixed critical systems, supporting both time- and event-triggered tasks and messages. For the schedule integration approach such a support of event-triggered systems could be realized by (1) defining suitable constraints for the time-triggered schedule synthesis, or by (2) applying a partition scheduling approach to determine suitable time-triggered cluster schedules, abstracting the event-triggered applications. For (1), a timing analysis of the event-triggered application allows to determine constraints for the schedule synthesis of the time-triggered schedule such that the idle-time is arranged to accommodate event-triggered applications, satisfying their timing constraints. For instance, the approaches and guidelines presented in [TSPS12, MSKS13] might form the basis of such an extension. For (2), the combination of partition scheduling with schedule integration might allow to efficiently integrate event-triggered applications with time-triggered applications while keeping the system highly modular. Essential for this approach will be to determine suitable constraints for the distributed partitions which are applied if a conflict refinement is necessary.

Schedule synthesis for variant management. Also for the multi-schedule synthesis presented in Chapter 5, various extension opportunities exist. In this thesis we focused on determining variant schedules such that the difference between variants is minimal. However, as the results show, this might lead to suboptimal variant schedules compared to generating each variant schedule individually. In future work, the multi-schedule synthesis framework might be extended to take additional design objectives such as the control performance of applications into account. The framework might then decide to define individual schedules to common parts in favor of an increased system performance. A multi-objective optimization might then allow to determine variant schedules for optimized system performance while the development cost is minimized due to variant-awareness.

Moreover, an important line of future work is also to investigate variant-aware event-triggered scheduling. Similar to the extension proposed for schedule integration, approaches from the domain of component-based design and partition scheduling might allow to abstract event-triggered tasks and messages as time-triggered components. These components could then be represented by processes in our system model and allow to apply our variant-aware schedule synthesis for event-triggered applications. Selecting suitable constraints for distributed applications, consisting of multiple components, will then be essential.

Bibliography

[ABI14] ABIresearch. Ethernet in-vehicle networking to feature in 40% of vehicles shipping globally by 2020, Jan. 2014. https://www.abiresearch.com/press/ethernet-in-vehicle-networking-to-feature-in-40-of/.

[ACPF14] A. Azim, G. Carvajal, R. Pellizzoni, and S. Fischmeister. Generation of communication schedules for multi-mode distributed real-time applications. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 1–6, 2014.

[ACZD94] G. Agrawal, B. Chen, W. Zhao, and S. Davari. Guaranteeing synchronous message deadlines with the timed token medium access control protocol. *IEEE Trans. on Computers*, 43(3):327–339, Mar. 1994.

[Alb04] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In *Proc. of Embedded World*, pages 235–252, 2004.

[ASA10] ASAM. ASAM MCD-2 NET FIBEX V3.1.1, 2010.

[ATB93] N. Audsley, K. Tindell, and A. Burns. The end of the line for static cyclic scheduling? In *Proc. of Euromicro Workshop on Real-Time Systems (EMWRT)*, pages 36–41, 1993.

[ATD+09] E. Armengaud, A. Tengg, M. Driussi, M. Karner, C. Steger, and R. Weiss. Automotive software architecture: Migration challenges from an event-triggered to a time-triggered communication scheme. In *Proc. of Workshop on Intelligent solutions in Embedded Systems (WISES)*, pages 95–103, 2009.

[AUT14] AUTOSAR. AUTOSAR 4.2.1, Oct. 2014.

[BÖ7] C. Böke. FlexRay is Driving. *Vector Informatics*, Mar. 2007.

[BCG12] M. Broy, M. V. Cengarle, and E. Geisberger. Cyber-physical systems: Imminent challenges. In *Large-Scale Complex IT Systems. Development, Operation and Management*, volume 7539 of *Lecture Notes in Computer Science*, pages 1–28. Springer Berlin Heidelberg, 2012.

[BD07] T. Braun and F. Deubzer. New variant management using multiple-domain mapping. In *Proc. of Design Structure Matrix Conference (DSM)*, pages 363–372, 2007.

[Ben96] A. Bender. Design of an optimal loosely coupled heterogeneous multiprocessor system. In *Proc. of European Design and Test Conference (ED&TC)*, pages 275–281, 1996.

[BFSS08] A. Belloni, R. Freund, M. Selove, and D. Simester. Optimizing product line designs: Efficient methods and comparisons. *Management Science*, 54(9):1544–1552, Jul. 2008.

[BGG+14] C. Buckl, M. Geisinger, D. Gulati, F. J. Ruiz-Bertol, and A. Knoll. CHROMOSOME: A run-time environment for plug & play-capable embedded real-time systems. *ACM SIGBED Reviews*, 11(3):36–39, Nov. 2014.

[BJM97] P. Bjorn-Jorgensen and J. Madsen. Critical path driven cosynthesis for heterogeneous target architectures. In *Proc. of Hardware/Software Codesign (CODES/-CASHE)*, pages 15–19, 1997.

[BKPS07] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proc. of the IEEE*, 95(2):356–373, Feb. 2007.

[BMW09] BMW Group. *Der neue BMW 7er*. Vieweg+Teubner, 2009.

[Bor14] K. Borgeest. *Elektronik in der Fahrzeugtechnik*. ATZ/MTZ-Fachbuch. Vieweg+Teubner Verlag, 2014.

[BPG00] J. Berwanger, M. Peller, and R. Grießbach. Byteflight — A new protocol for safety critical applications. In *Proc. of FISITA World Automotive Congress (FISITA)*, 2000.

[BPS08] J. Berwanger, M. Peteratzinger, and A. Schedl. FlexRay startet durch - FlexRay-Bordnetz für Fahrdynamik und Fahrerassistenzsysteme. *Elektronik automotive: Sonderausgabe 7er BMW*, 2008.

[Bro06] M. Broy. Challenges in automotive software engineering. In *Proc. of International Conference on Software Engineering (ICSE)*, pages 33–42, 2006.

[CAN06] ISO 11898 Road vehicles – Controller Area Network (CAN) – Part 3: Low-speed, fault-tolerant, medium-dependent interface, 2006.

[CAN13] ISO 11898 Road vehicles – Controller Area Network (CAN) – Part 2,4-6: High-speed medium access unit, 2003-2013.

[Cap13] J. Capparella. Audi adding 11 models to expand lineup by 2020. *Automobile Magazine*, Dec. 2013.

[Cen06] A. Cena, G .and Valenzano. On the properties of the flexible time division multiple access technique. *IEEE Trans. on Industrial Informatics*, 2(2):86–94, May 2006.

[Cha09] R. N. Charette. This car runs on code. *IEEE Spectrum*, 46(3):3, Feb. 2009.

[Chi07] J. W. Chinneck. *Feasibility and Infeasibility in Optimization:: Algorithms and Computational Methods*, volume 118. Springer, 2007.

[CJG72] E. G. Coffman Jr. and R. L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[CV04] G. Cena and A. Valenzano. Performance analysis of Byteflight networks. In *Proc. of Workshop on Factory Communication Systems (WFCS)*, pages 157–166, 2004.

[DB05] R. I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 389–398, 2005.

[DBBL07] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.

[DH14] J. Dvorak and Z. Hanzalek. Multi-variant time constrained FlexRay static segment scheduling. In *Proc. of Workshop on Factory Communication Systems (WFCS)*, pages 1–8, 2014.

[Din10] S. Ding. Scheduling approach for static segment using hybrid genetic algorithm in FlexRay systems. In *Proc. of Computer and Information Technology (CIT)*, pages 2355–2360, 2010.

[DK14] A. Darbandi and M. K. Kim. Schedule optimization of static messages with precedence relations in FlexRay. In *Proc. of International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 495–500, 2014.

[DKK14] A. Darbandi, S. Kwon, and M. K. Kim. Scheduling of time triggered messages in static segment of FlexRay. *International Journal of Software Engineering and Its Applications*, 8(6):195–208, 2014.

[DMTT05] S. Ding, N. Murakami, H. Tomiyama, and H. Takada. A GA-based scheduling method for FlexRay systems. In *Proc. of Embedded Software (EMSOFT)*, May 2005.

[DP94] M. E. Dalkilic and V. Pitchumani. A multi-schedule approach to high-level synthesis. In *Proc. of International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, pages 572–575, 1994.

[DRW98] R. P. Dick, D. L. Rhodes, and W. Wolf. TGFF: task graphs for free. In *Proc. of International Workshop on Hardware/Software Codesign (CODES/CASHE)*, pages 97–101, 1998.

[DTT08] S. Ding, H. Tomiyama, and H. Takada. An Effective GA-based Scheduling Algorithm for FlexRay Systems. *IEICE Trans. on Information and Systems*, 91(8):2115–2123, Aug. 2008.

[EB10] P. Emberson and I. Bate. Stressing search with scenarios for flexible solutions to real-time task allocation problems. *IEEE Trans. on Software Engineering*, 36(5):704–718, Sep. 2010.

[EDPP00] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 8(5):472–491, Oct. 2000.

[ETAa] ETAS. Ascet. http://www.etas.com/en/products/ascet_software_products.php.

[ETAb] ETAS. RTA-OSEK. http://www.etas.com/en/products/rta_osek.php.

[Eve11] S. Even. *Graph algorithms*. Cambridge University Press, 2011.

[Fea04] P. Feautrier. Scalable and modular scheduling. In *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 433–442. Springer Berlin Heidelberg, 2004.

[Fea06] P. Feautrier. Scalable and structured scheduling. *International Journal of Parallel Programming*, 34(5):459–487, 2006.

[FH04] C. Ferdinand and R. Heckmann. aiT: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 377–383. Springer US, 2004.

[Fle10] FlexRay Consortium. FlexRay communications system protocol specification version 3.0.1, 2010. http://www.flexray.com.

[Fle13] Road vehicles – FlexRay communications system Part 1-5, 2013.

[GGTL14] S. Graf, M. Glaß, J. Teich, and C. Lauer. Multi-variant-based design space exploration for automotive embedded systems. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 1–6, 2014.

[GGW+13] S. Graf, M. Glass, D. Wintermann, J. Teich, and C. Lauer. Ivam: Implicit variant modeling and management for automotive embedded systems. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, 2013.

[GHN08] M. Grenier, L. Havet, and N. Navet. Configuring the communication on FlexRay: the case of the static segment. In *Proc. of Embedded Real Time Software (ERTS)*, 2008.

[GLSC12] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 1227–1232, 2012.

[GN02] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proc. of the National Academy of Sciences*, 99(12):7821–7826, 2002.

[Grz11] A. Grzemba. *MOST: The automotive multimedia network, from MOST25 to MOST150*. Franzis, 2011.

[Har12] F. Hartwich. CAN with flexible data-rate. In *Proc. of International CAN Conference (iCC2012)*, 2012.

[HBC+07] A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, and S. Ramesh. Performance analysis of FlexRay-based ECU networks. In *Proc. of Design Automation Conference (DAC)*, 2007.

[HE05] A. Hamann and R. Ernst. TDMA time slot and turn optimization with evolutionary search techniques. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 312–317, 2005.

[HLW+14] M. Hu, J. Luo, Y. Wang, M. Lukasiewycz, and Z. Zeng. Holistic scheduling of real-time applications in time-triggered in-vehicle networks. *IEEE Trans. on Industrial Informatics*, 10(3):1817–1828, Aug. 2014.

[IEE15] IEEE. Time-Sensitive Networking Task Group, 2015. http://www.ieee802.org/1/pages/tsn.html.

[ILO] ILOG. CPLEX. http://www.ilog.com/products/cplex/.

[IO98] M. Iverson and F. Ozguner. Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment. In *Proc. of Heterogeneous Computing Workshop (HCW)*, pages 70–78, 1998.

[ISO12] Road vehicles – Functional safety Part 1-10, 2012.

[KB03] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, Jan. 2003.

[KCBQ14] C. M. Kozierok, C. Correra, R. B. Boatright, and J. Quesnelle. *Automotive Ethernet The Definitive Guide*. Intrepid Control Systems, 2014.

[KG94] H. Kopetz and G. Grunsteidl. TTP - a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan. 1994.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220, 4598:671–680, 1983.

[Kop91] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 86–101. Springer Berlin Heidelberg, 1991.

[Kop95] H. Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In *Proc. of Future Trends of Distributed Computing Systems (FTDCS)*, pages 2–9, 1995.

[KP08] J. Koetz and S. Poledna. Making FlexRay a reality in a premium car. In *Proc. of SAE Convergence*, 2008.

[KPJ13] M. Kang, K. Park, and M.-K. Jeong. Frame packing for minimizing the bandwidth consumption of the FlexRay static segment. *IEEE Trans. on Industrial Electronics*, 60(9):4001–4008, Sep. 2013.

[KWL+12] J. Kim, K. We, C.-G. Lee, K.-J. Lin, and Y. S. Lee. HW resource componentizing for smooth migration from single-function ECU to multi-function ECU. In *Proc. of Symposium on Applied Computing (SAC)*, pages 1821–1828, 2012.

[Lee09] S. Leef. AUTOSAR and FlexRay: A tale of two standards. Technical report, Mentor Graphics, 2009. http://www.hanser-automotive.de/uploads/media/AUTOSAR_and_FlexRay.pdf.

[Lev73] G. Levi. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO*, 9(4):341–352, 1973.

[LGRT11] M. Lukasiewycz, M. Glaß, F. Reimann, and J. Teich. Opt4J - a modular framework for meta-heuristic optimization. In *Proc. of Genetic and Evolutionary Computing Conference (GECCO 2011)*, pages 1723–1730, 2011.

[LGTM09] M. Lukasiewycz, M. Glaß, J. Teich, and P. Milbredt. FlexRay schedule optimization of the static segment. In *Proc. of Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2009.

[LINar] ISO 17987 - Road vehicles – Local Interconnect Network (LIN) Part 1-7, To appear.

[LKY+00] Y.-H. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Proc. of Dependable Systems and Networks (DSN)*, pages 14–23, 2000.

[LLP97] C.-Y. Lee, L. Lei, and M. Pinedo. Current trends in deterministic scheduling. *Annals of Operations Research*, 70(0):1–41, 1997.

[LMV02] A. Lodi, S. Martello, and D. Vigo. Recent advances on two-dimensional bin packing problems. *Discrete Applied Mathematics*, 123(1-3):379–396, Nov. 2002.

[LSGC12] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular scheduling of distributed heterogeneous time-triggered automotive systems. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 665–670, 2012.

[Lup12] M. Lupa. 7 questions on MQB. *Volkswagen Das Auto. Magazine*, 2012.

[Mat15] MathWorks Inc. Simulink 8.5, 2015. http://www.mathworks.com/products/simulink/.

[MB08] L. Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340. Springer Berlin Heidelberg, 2008.

[Men10] Mentor Graphics. Top-down design of distributed embedded systems in light of timing considerations. *Technical Paper*, 2010.

[Mic96] Z. Michalewicz. *Genetic Algorithms + Data Structures= Evolution Programs*. Springer, 1996.

[MSKS13] P. Meyer, T. Steinbach, F. Korf, and T. Schmidt. Extending IEEE 802.1 AVB with time-triggered scheduling: A simulation study of the coexistence of synchronous and asynchronous traffic. In *Proc. of Vehicular Networking Conference (VNC)*, pages 47–54, 2013.

[MSR13] C. Manz, M. Stupperich, and M. Reichert. Towards integrated variant management in global software engineering: An experience report. In *Proc. of International Conference on Global Software Engineering (ICGSE)*, pages 168–172, 2013.

[MSS89] H. Matsuo, J. C. Suh, and R. S. Sullivan. A controlled search simulated annealing method for the single machine weighted tardiness problem. *Annals of Operations Research*, 21(1):85–108, 1989.

[NGA09] V. Nelis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *Proc. of Euromicro Conference on Real-Time Systems (ECRTS)*, pages 151–160, 2009.

[NNS+11] M. Negrean, M. Neukirchner, S. Stein, S. Schliecker, and R. Ernst. Bounding mode change transition latencies for multi-mode real-time distributed applications. In *Proc. of Emerging Technologies & Factory Automation (ETFA)*, pages 1–10, 2011.

[NPP01] S. A. Nelson, M. B. Parkinson, and P. Y. Papalambros. Multicriteria optimization in product platform design. *Journal of Mechanical Design*, 123(2):199–204, 2001.

[Obe11] R. Obermaisser. *Time-triggered communication*. CRC Press, Inc., 2011.

[Ope] OpenDSE. Open design space exploration framework. http://opendse.sf.net/.

[Ope15] OPEN Alliance Special Interest Group, 2015. http://www.opensig.org/.

[OSE06] ISO 17356 - Road vehicles – Open interface for embedded automotive applications – Part 1-5, 2006.

[PCFW05] G. Pardo-Castellote, B. Farabaugh, and R. Warren. An introduction to DDS and data-centric communications, 2005. http://omg.org/news/whitepapers/Intro_To_DDS.pdf.

[PEP04] P. Pop, P. Eles, and Z. Peng. Schedulability-driven communication synthesis for time triggered embedded systems. *Real-Time Systems*, 26(3):297–325, 2004.

[PEP05] P. Pop, P. Eles, and Z. Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *ACM Trans. on Embedded Computing Systems*, 4(1):112–140, Feb. 2005.

[PEPP01] P. Pop, P. Eles, T. Pop, and Z. Peng. An approach to incremental design of distributed embedded systems. In *Proc. of Design Automation Conference (DAC)*, pages 450–455, 2001.

[PEPP04] P. Pop, P. Eles, Z. Peng, and T. Pop. Scheduling and mapping in an incremental design methodology for distributed real-time embedded systems. *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 12(8):793–811, Aug. 2004.

[PPE+08] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39:205–235, 2008.

[PPEP07] T. Pop, P. Pop, P. Eles, and Z. Peng. Bus access optimisation for FlexRay-based distributed embedded systems. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 1–6, 2007.

[Pri92] P. Prisaznuk. Integrated modular avionics. In *Proc. of National Aerospace and Electronics Conference (NAECON)*, pages 39–45 vol.1, 1992.

[PRO] PROFIBUS & PROFINET International. Profinet. http://www.profibus.com/technology/profibus/.

[PS11] I. Park and M. Sunwoo. FlexRay network parameter optimization method for automotive applications. *IEEE Trans. on Industrial Electronics*, 58(4):1449–1459, Apr. 2011.

[pur06] pure-systems GmbH. Variant management with pure::variants. *Technical White Paper*, 2006.

[Rei11] K. Reif. *Bosch Autoelektrik und Autoelektronik*. Vieweg+Teubner, 2011.

[Rei14] K. Reif. *Automobilelektronik*. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2014.

[Rob14] Robert Bosch GmbH. *Bosch Automotive Electrics and Automotive Electronics*. Bosch Professional Automotive Information. Springer Fachmedien Wiesbaden, 2014.

[SAW+14] F. Sagstetter, S. Andalam, P. Waszecki, M. Lukasiewycz, H. Staehle, S. Chakraborty, and A. Knoll. Schedule integration framework for time-triggered automotive architectures. In *Proc. of Design Automation Conference (DAC)*, pages 1–6, 2014.

[SCTQ09] V. Schulte-Coerne, A. Thums, and J. Quante. Automotive software: Characteristics and reengineering challenges. In *Softwaretechnik-Trends*, volume 29. 2009.

[SGC+11] R. Schneider, D. Goswami, S. Chakraborty, U. Bordoloi, P. Eles, and Z. Peng. On the quantification of sustainability and extensibility of FlexRay schedules. In *Proc. of Design Automation Conference (DAC)*, pages 375–380, 2011.

[SL03] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 2–13, 2003.

[SL08] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. on Embedded Computing Systems*, 7(3):30:1–30:39, Apr. 2008.

[SLC] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty. Generalized asynchronous time-triggered scheduling for flexray. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*. to appear.

[SLC13] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty. Schedule integration for time-triggered systems. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 52–58, 2013.

[SN13] J. Sobotka and J. Novak. Automation of automotive integration testing process. In *Proc. of Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, pages 349–352, 2013.

[SPT09] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or EDF scheduling. In *Proc. of Design, Automation and Test in Europe (DATE)*, pages 99–104, 2009.

[SRN+09] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):979–992, Jul. 2009.

[SS07] K. Schmidt and E. Schmidt. Systematic message schedule construction for Time-Triggered CAN. *IEEE Trans. on Vehicular Technology*, 56(6):3431–3441, Nov 2007.

[SS09a] E. Schmidt and K. Schmidt. Message scheduling for the FlexRay protocol: The dynamic segment. *IEEE Trans. on Vehicular Technology*, 58(5):2160–2169, Jun. 2009.

[SS09b] K. Schmidt and E. Schmidt. Message scheduling for the FlexRay protocol: The static segment. *IEEE Trans. on Vehicular Technology*, 58(5):2170–2179, Jun. 2009.

[ST12] T. Streichert and M. Traub. *Elektrik/Elektronik-Architekturen im Kraftfahrzeug: Modellierung und Bewertung von Echtzeitsystemen*. Springer-Verlag, 2012.

[Ste10] W. Steiner. An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks. In *Proc. of Real-Time Systems Symposium (RTSS)*, pages 375–384, 2010.

[Ste14] W. Steiner. Echtzeitanwendungen mit Automotive Ethernet. Sep. 2014. http://www.elektroniknet.de/automotive/bussysteme/artikel/113002/.

[SVG11] T. Schenkelaars, B. Vermeulen, and K. Goossens. Optimal scheduling of switched FlexRay networks. In *Proc. of Design, Automation and Test in Europe (DATE)*, 2011.

[SWS+16] F. Sagstetter, P. Waszecki, S. Steinhorst, M. Lukasiewycz, and S. Chakraborty. Multi-schedule synthesis for variant management in automotive time-triggered systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 35(4):637–650, Apr. 2016.

[Sym15] Symtavision. Symta/s 3.7, 2015. https://www.symtavision.com/products/symtas-traceanalyzer/.

[SYP+09] S. Samii, Y. Yin, Z. Peng, P. Eles, and Y. Zhang. Immune genetic algorithms for optimization of task priorities and FlexRay frame identifiers. In *Proc. of Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 486–493, 2009.

[SZ13] J. Schäuffele and T. Zurawka. *Automotive Software Engineering: Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen*. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2013.

[TBW95] K. Tindell, A. Burns, and A. Wellings. Calculating Controller Area Network (CAN) message response times. *Control Engineering Practice*, 3(8):1163 – 1169, 1995.

[TC94] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2–3):117 – 134, 1994. Parallel Processing in Embedded Real-time Systems.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Proc. of International Symposium on Circuits and Systems (ISCAS)*, pages 101–104, 2000.

[TDH11] J. Thomas, C. Dziobek, and B. Hedenetz. Variability management in the AUTOSAR-based development of applications for in-vehicle systems. In *Proc. of Variability Modeling of Software-Intensive Systems (VaMos)*, pages 137–140, 2011.

[TH02] S. Thiel and A. Hein. Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72, Jul. 2002.

[THW02] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. on Parallel and Distributed Systems*, 13(3):260–274, Mar. 2002.

[TSP11] D. Tamas-Selicean and P. Pop. Optimization of time-partitions for mixed-criticality real-time distributed embedded systems. In *Proc. of International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 1–10, 2011.

[TSPS12] D. Tamas-Selicean, P. Pop, and W. Steiner. Synthesis of communication schedules for TTEthernet-based mixed-criticality systems. In *Proc. of Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 473–482, 2012.

[TTA08] TTA Group. TTEthernet Specification, 2008. http://www.ttagroup.org.

[TTC04] ISO 11898 Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication, 2004.

[Vec] Vector. osCAN - real time operating system based on the OSEK/VDX standard. http://vector.com/vi_oscan_en.html.

[Vec08] Vector. From pilot studies to production development. *Technical Article*, 2008.

[Vec15] Vector. CANoe 8.5, 2015. http://vector.com/vi_canoe_en.html.

[Völ13] L. Völker. SOME/IP – Die Middleware für Ethernet-basierte Kommunikation. *Hanser automotive networks*, Nov. 2013. http://www.hanser-automotive.de/printarchiv/article/someip-die-middleware-fuer-ethernetbasierte-kommunikation.html.

[VSE09] S. Voss, M. Sorea, and K. Echtle. SAL-based symbolic scheduling in time-triggered networks. In *Integrated Formal Methods*, volume 5423, pages 200–214. Springer Berlin Heidelberg, 2009.

[WJP+05] Z. Wei, C. Jike, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *Proc. of Application of Concurrency to System Design (ACSD)*, pages 132–141, 2005.

[WLMC13] P. Waszecki, M. Lukasiewycz, A. Masrur, and S. Chakraborty. How to engineer tool-chains for automotive E/E architectures? *ACM SIGBED Review*, 10(4):6–15, Dec. 2013.

[WT06] E. Wandeler and L. Thiele. Interface-based design of real-time systems with hierarchical scheduling. In *Proc. of Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 243–252, 2006.

[WZ15] V. Woollaston and E. Zolfagharidard. Tesla's 'Insane Mode' just got FASTER: Elon Musk reveals over-the-air software upgrade that will boost acceleration even more. *dailymail.co.uk*, Jan. 2015. http://www.dailymail.co.uk/sciencetech/article-2932307/Tesla-s-insane-mode-just-got-FASTER-Elon-Musk-reveals-air-software-upgrade-boost-acceleration-more.html.

[YM09] Y. Yamaguchi and T. Murakami. Adaptive control for virtual steering characteristics on electric vehicle using steer-by-wire system. *IEEE Trans. on Industrial Electronics*, 56(5):1585–1594, May 2009.

[ZDGSV11] H. Zeng, M. Di Natale, A. Ghosal, and A. Sangiovanni-Vincentelli. Schedule optimization of time-triggered systems communicating over the FlexRay static segment. *IEEE Trans. on Industrial Informatics*, 7(1):1–17, Feb. 2011.

[ZS06] H. Zhao and R. Sakellariou. Scheduling multiple DAGs onto heterogeneous systems. In *Proc. of International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[ZS14] W. Zimmermann and R. Schmidgall. *Bussysteme in der Fahrzeugtechnik*. ATZ/MTZ-Fachbuch. Springer Fachmedien Wiesbaden, 2014.

[ZYN+10] Q. Zhu, Y. Yang, M. Natale, E. Scholte, and A. Sangiovanni-Vincentelli. Optimizing the software architecture for extensibility in hard real-time distributed systems. *IEEE Trans. on Industrial Informatics*, 6(4):621–636, Nov. 2010.

[ZZD+09] H. Zeng, W. Zheng, M. Di Natale, A. Ghosal, P. Giusto, and A. Sangiovanni-Vincentelli. Scheduling the FlexRay bus using optimization techniques. In *Proc. of Design Automation Conference (DAC)*, pages 874–877, 2009.

List of Tables

List of Figures

List of Acronyms

ABS	Antilock Braking System
ACC	Adaptive Cruise Control
ASW	Application Software
AUTOSAR	AUTomotive Open System ARchitecture
BSW	Basic Software
CAN	Controller Area Network
CAN FD	CAN with Flexible Data-Rate
CSMA/CA	Carrier Sense Multiple Access/Collision Avoidance
DAG	Directed Acyclic Graph
DSE	Design Space Exploration
E/E	Electrical/Electronic
EA	Evolutionary Algorithm
ECU	Electronic Control Unit
EDF	Earliest Deadline First
ESC	Electronic Stability Control
FIBEX	FIeld Bus Exchange Format
FP	Fixed Priority
FTDMA	Flexible Time Division Multiple Access
GA	Genetic Algorithm
IIS	Irreducible Inconsistent Set

ILP	Integer Linear Programming
IP	Internet Protocol
IPv6	Internet Protocol version 6
LIN	Local Interconnect Network
MOST	Media Oriented Systems Transport
OSEK/VDX	Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug/Vehicle Distributed Executive
OTA	Over-The-Air programming
RM	Rate-Monotonic
RTC	Real-Time Calculus
RTE	Real-Time Environment
RTOS	Real-Time Operating System
SA	Simulated Annealing
SMT	Satisfiability Modulo Theories
SWC	Software Component
SymTA/S	Symbolic Timing Analysis for Systems
TCP	Transmission Control Protocol
TCS	Traction Control System
TDMA	Time Division Multiple Access
TSN	Time Sensitive Networking
TTCAN	Time-Triggered-CAN
TTP	Time-Triggered Protocol
UDP	User Datagram Protocol
VFB	Virtual Function Bus
WCET	Worst Case Execution Time