Schedule Integration Framework for Time-Triggered Automotive Architectures

Florian Sagstetter, Sidharta Andalam,
Peter Waszecki, Martin Lukasiewycz
TUM CREATE, Singapore
florian.sagstetter@tum-create.edu.sg

Hauke Stähle,
Samarjit Chakraborty, Alois Knoll
TU Munich, Germany
samarjit@tum.de

ABSTRACT

Automotive Electrical/Electronic (E/E)-architectures consist of various components which are generally developed independently. Due to the increasing size and complexity, component integration is highly challenging and already slight modifications to components or subsystems often require expensive re-testing and re-validation. As a remedy, we propose a framework for modular architectures based on a data-centric description and a fully time-triggered scheduling. This modular design approach is enabled by a novel methodology for schedule integration where local schedules are defined independently for subsystems before being integrated into a global schedule. This divide-and-conquer approach significantly reduces the integration complexity while the system becomes highly composable. Our experimental results give evidence of the efficiency and versatility of the proposed approach, using networks based on a time-triggered automotive Ethernet.

1. INTRODUCTION

Innovation in the automotive industry today is largely driven by software. However, as novel functions are often developed by suppliers, they are commonly integrated as additional (hardware) components. Hence, the actual software-based functionality is already implemented on respective Electronic Control Units (ECUs) when provided to the car manufacturer. During the integration phase, the different components are then considered from a *black-box* perspective. Each component is defined by its messages together with some semi-formal description of its behavior and constraints. Intensive testing and analysis is required to prevent *unintentional feature interaction* and ensure correct functionality. Due to the rapidly growing complexity of this integration process, the current design approach is reaching its limits [1]. At the same time, the design paradigm in the automotive industry is currently shifting from event-triggered to time-triggered systems, e.g., through the introduction of automotive Ethernet. Therefore, a fundamental paradigm change in the design of automotive Electrical/Electronic (E/E)-architectures is required.

Related work. The AUTomotive Open System ARchitecture (AUTOSAR) partnership is one of the efforts undertaken to define an open and uniform software platform to improve the portability of software [2]. We assume such a homogeneous software platform and extend it with a data-centric description and a modular design process. In particular, here we focus on a modular scheduling approach based on schedule integration. 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 [3][4]. Allowing to apply different scheduling strategies like Earliest Deadline First (EDF) or Rate-Monotonic (RM) scheduling for individual schedulers. However, component-based scheduling is not applicable to automotive architectures which implement distributed applications running on several networked resources. In contrast, our paper considers a directed acyclic graph model where each node represents a task and the transitions represent the dependencies between tasks.

Our framework is based on a fully time-triggered system, providing the basis for temporal composablity [5][6] which is an asset in the automotive domain. Time-triggered systems are increasingly used in E/E-architectures, in particular, in the form of FlexRay [7] and upcoming automotive Ethernet [8]. Scheduling of time-triggered systems is a challenging task and various approaches have been proposed. In [9], an approach using the model-checker *SAL* to determine a system schedule is presented. An Satisfiability Modulo Theories (SMT)-based approach to generate time-triggered schedules for *TTEthernet* is presented in [10]. Finally, two Integer Linear Programming (ILP)-based approaches have been presented for the *FlexRay* bus in [7][11]. While [7] addresses the problem at *job-level*, [11] applies the schedule optimization at *task-level*. Most of these approaches perform well for small subsystem schedules, but do not scale well for larger systems.

Contributions of the paper. We propose a framework for scheduling modular time-triggered systems in the automotive domain which is well suited for solving large and complex scheduling problems. It is based on a data-centric description which allows to decouple the design process of individual software components. A modular scheduling approach allows to add or update applications and subsystems incrementally. We present a schedule integration approach that combines independent subsystem schedules into a global schedule. While this is of great importance in the automotive domain, schedule integration has not been sufficiently studied in scientific literature. It was first addressed in [12] with a focus on FlexRay. In the work at hand, we extend this approach with a concurrent task and message scheduling and generalize it to support time-triggered Ethernet. An extended conflict refinement allows to adapt predefined subsystem schedules when required.

The focus of this paper is not to obtain a globally optimal schedule, but rather to integrate locally optimized application configurations according to their specific requirements. The

DAC '14 June 01 - 05 2014, San Francisco, CA, USA

ACM 978-1-4503-2730-5/14/06...$15.00.
http://dx.doi.org/10.1145/2593069.2593211.

This publication is made possible by the Singapore National Research Foundation under its Campus for Research Excellence And Technological Enterprise (CREATE) programme.

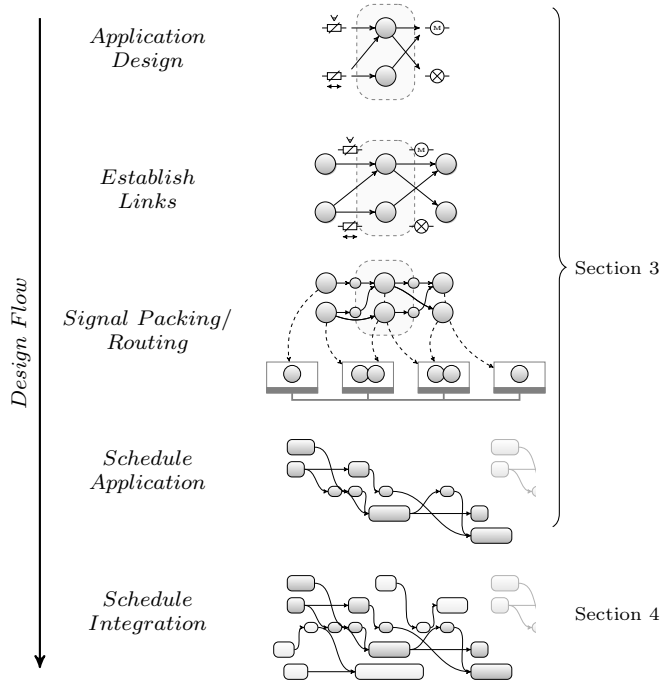


Figure 1: Design flow of our framework from a data-centric application definition to a fully configured system, including a global schedule.

presented schedule integration does not only enable the design of highly composable architectures, it also significantly improves the runtime of concurrent task and message scheduling compared to existing work.

Paper outline. Section 2 introduces our framework. Section 3 presents the data-centric design flow to generate independent schedules for subsystems. Section 4 then presents our schedule integration approach to integrate subsystem schedules into a global schedule. Finally, Section 5 presents test cases and a case study for architectures based on time-triggered automotive Ethernet, before Section 6 concludes the paper.

2. FRAMEWORK

The proposed framework is illustrated in Figure 1. It is based on a data-centric design and a fully 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. (3) Based on an implicitly defined task mapping, a signal packing and routing is defined, introducing messages for task communication in distributed applications. (4) A schedule is created, specifying the exact starting times for all tasks and messages. (5) We apply our schedule integration approach to integrate the individual subsystem schedules into a global schedule. It is based on a fully time-triggered system where all tasks and messages are scheduled by a global clock. For time-triggered systems, properties like worst-case execution time and maximal end-to-end delay are commonly taken into account during the design phase. Hence, the testing and analysis required to validate correct system behavior is significantly reduced. On the other hand, the design complexity for time-triggered systems typically grows exponentially if a system schedule is generated from the scratch, see [9][11]. As a remedy, our schedule integration exploits the predictability of time-triggered systems. It allows to add or update application or subsystem schedules in an iterative process, measurably reducing the integration efforts. Furthermore, it allows to optimize a subsystem configuration according to its individual requirements, improving the application performance. The only requirements are a homogeneous software platform like an AUTOSAR based operating system, and a time-triggered architecture as supported by automotive Ethernet used in next-generation vehicles.

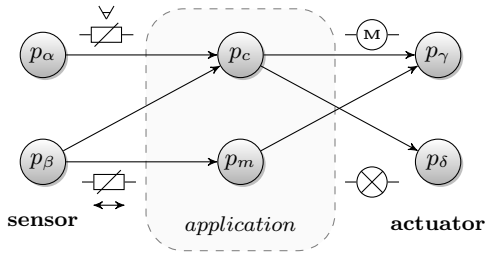


Figure 2: Task graph for a data-centric steer-by-wire application. The application subscribes to two sensor topics (,) to calculate the steering angle published to the actuator topic ( ). An indicator topic ( ) allows indicating errors. The application tasks p_c and p_m only have data but no hardware dependencies.

3. SUBSYSTEM DESIGN

This section illustrates the design process from an application developed according to data-centric design principles to a configured subsystem accessing hardware resources.

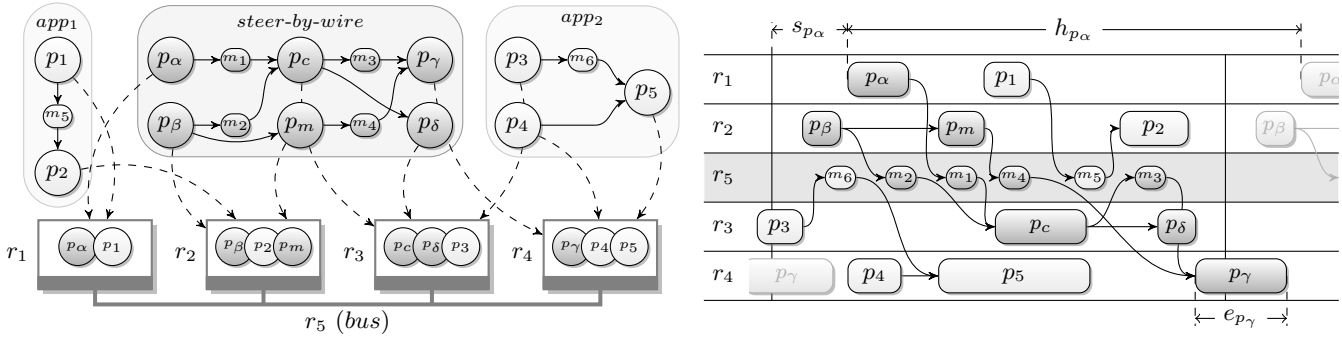
3.1 Data-Centric Design

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 2 shows the resulting task graph for a *steer-by-wire* application after the links between application and sensor and actuator tasks have been established. As tasks abstracting hardware peripherals are explicitly bound to a particular hardware resource, an implicit task mapping is given. For applications distributed over several network nodes, additional messages are introduced if required. Figure 3(a) illustrates the final task graph Π_F of the application F after signal packing and message routing.

3.2 Generate Subsystem Schedules

To generate a subsystem schedule, for each task or message p a start time $\mathbf{s_p}$ has to be determined. The data-centric description of each application F defines the processing time e_p



(a) Application task graphs and their mapping to the architecture

(b) Schedule defining start times of each process

Figure 3: (a) Task graphs of three applications are mapped to four ECUs connected by a communication bus. If the tasks of an application are distributed over different resources, additional messages are introduced for communication. The dark shaded task graph (◯) represents the application from Figure 2 after signal packing and routing. (b) Potential schedule, defining start times $\mathbf{s_p}$ for each task or message.

and period h_p for each $p \in F$. Furthermore, after signaling and routing, for each process p a mapping to an ECU or bus r is defined. Based on these parameters, a system schedule is determined. Figure 3 illustrates the task mapping of three applications to an architecture and a potential schedule for each application. The scheduling algorithm determines 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, we define the following function:

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

Without loss of generality, we assume non-preemptive time-triggered scheduling such that a resource r always completes the execution of one process before another process is started. We denote the set of processes on a specific resource r as P_r. To create a feasible schedule, two processes $p, \tilde{p} \in P_r$ must not use r at the same point in time $\mathbf{t}$:
$\forall r \in R, \mathbf{t} \in \mathbb{R}^+, p, \tilde{p} \in P_r, p \neq \tilde{p}$:

$$r(p,\mathbf{t}) + r(\tilde{p},\mathbf{t}) \leq 1 \tag{2}$$

To determine the sub system schedules, we use an SMT approach which performs a concurrent task and message scheduling, based on a previously presented ILP approach. As the focus here lies on schedule integration, please refer to [11] for the ILP formulation to determine a feasible schedule. Depending on the application requirements, an optimization objective might be defined for a minimal end-to-end delay [7], the control performance [13], or the extensibility to ease later schedule integration [14]. The determined subsystem or cluster schedules might then be combined to a global schedule by our schedule integration approach.

4. INTEGRATION

The schedule integration approach presented in this section forms the basis for a highly flexible integration process. First, we describe the schedule integration, defining a feasible offset for each cluster schedule. Second, a conflict refinement allowing to adapt incompatible cluster schedules is presented. For the sake of simplicity, we refer to both tasks and messages as *process p*, and to any subsystem as *cluster* in the following.

4.1 Schedule Integration

During schedule integration, independently create cluster schedules are combined to a global schedule. We define an offset $\mathbf{o_D}$ for each cluster schedule $D \in \mathcal{D}$ which allows to shift the whole cluster schedule by a constant time. This offset maintains the general structure of the cluster schedule and does not affect the subsystem behavior, as the start times of all processes within the cluster are adapted concurrently and the cluster schedule is executed periodically. Figure 3(b) shows a global schedule after integrating three subsystem schedules. In accordance with Constraint (2), two clusters D and $\tilde{D}$ can be integrated in a global schedule, if the following equation holds:
$\forall r \in R, \mathbf{t} \in \mathbb{R}^+, p \in D, \tilde{p} \in \tilde{D}$:

$$r(p, \mathbf{t} + \mathbf{o_D}) + r(\tilde{p}, \mathbf{t} + \mathbf{o_{\tilde{D}}}) \leq 1 \tag{3}$$

Hence, $\mathbf{o_D}$ and $\mathbf{o_{\tilde{D}}}$ must be selected such that no two processes $p \in D, \tilde{p} \in \tilde{D}$ intersect when scheduled on the same resource. As the cluster schedule is executed with the period $h_D = \underset{p \in D}{lcm}(h_p)$[1], $\mathbf{o_D}$ might have any value between 0 and h_D. To determine offsets for each cluster, we use a two step approach: First, we compute feasible intervals for the relative offset for each cluster pair $D, \tilde{D} \in \mathcal{D}$. The relative offset is defined as $\mathbf{o_D} - \mathbf{o_{\tilde{D}}}$, hence it defines the relation between the two offsets. Second, the final offsets for the whole set are determined with an SMT. Based on Equation (3), the intervals defining all feasible offsets for a single resource r are computed as follows:

$$\Delta(r, D, \tilde{D}) = \{x | x = \mathbf{o_D} - \mathbf{o_{\tilde{D}}}, \mathbf{o_D} \in [0, h_D], \mathbf{o_{\tilde{D}}} \in [0, h_{\tilde{D}}], \\ p \in D, \tilde{p} \in \tilde{D}, \exists \mathbf{t} : r(p, \mathbf{t} + \mathbf{o_D}) + r(\tilde{p}, \mathbf{t} + \mathbf{o_{\tilde{D}}}) \leq 1\} \tag{4}$$

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 defining the set of resources used by cluster D, we determine $I_{D,\tilde{D}}$, defining all feasible intervals $\lambda_{D,\tilde{D}} = [\lambda_{min}, \lambda_{max}]$ for the relative cluster offset for D, $\tilde{D}$:

$$I_{D,\tilde{D}} = \bigcap_{r \in R_D \cap R_{\tilde{D}}} \Delta(r, D, \tilde{D}) \tag{5}$$

Based on these intevals, we apply the following SMT formulation to determine the cluster offsets o_D:
$\forall D \in \mathcal{D}$:

$$0 \leq \mathbf{o_D} < h_D \tag{6}$$

$\forall D, \tilde{D} \in \mathcal{D}, D \neq \tilde{D}, \lambda_{D,\tilde{D}} \in I_{D,\tilde{D}}$:

$$\lambda_{min} \leq \mathbf{o_D} - \mathbf{o_{\tilde{D}}} \leq \lambda_{max} \tag{7}$$

Constraint (6) defines the boundaries for $\mathbf{o_D}$, while Constraint (7) ensures that all determined $\mathbf{o_D}$ lie in the previously determined intervals. Updating the process start-times for each cluster with the obtained offsets leads to the global system schedule.

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

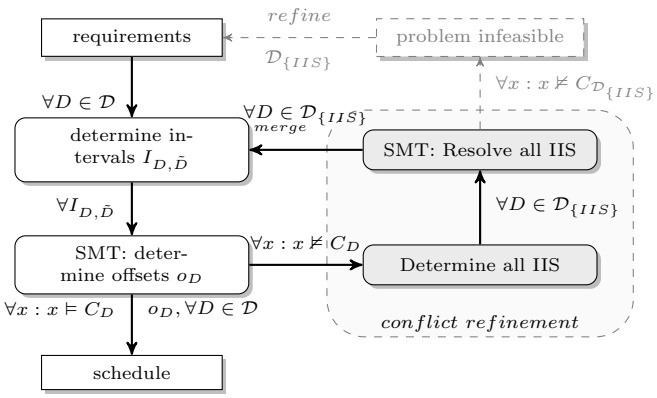


Figure 4: Schedule Integration flow. If no solution exists for the schedule integration ($\forall x : x \nvDash C_D$) a conflict refinement is applied to resolve infeasibilities in conflicting clusters. In case the conflict refinement fails ($\forall x : x \nvDash C_{\mathcal{D}_{\{IIS\}}}$), it provides information to refine the system configuration.

4.2 Conflict Refinement

The schedule integration approach adapts the start times s_p of all processes $p \in D$ for a cluster D collectively with the offset $\mathbf{o_D}$. However, as the general structure for each cluster $D \in \mathcal{D}$ is fixed, no feasible schedule x might exist such that all constraints C_D, defining the offset boundaries, are satisfied ($\forall x : x \vDash C_D$). In case no feasible solution exists for the initial set of clusters, a conflict refinement is required. It adapts the individual process start times without affecting the end-to-end timing delay defined in the cluster schedule. Hence, during conflict refinement the start-times of intermediate tasks within a task graph might be adapted such that the end-to-end timing delay defined during cluster schedule generation is not affected. For source or sink processes we assume that a decreasing end-to-end delay has no negative impact on the functionality, hence our algorithm might reduce the end-to-end delay but not increase it. Therefore, while the conflict refinement modifies the timing behavior of cluster schedules, all previously defined constrains are still valid. Figure 4 illustrates the schedule integration module of our framework. If no feasible solution exists ($\forall x : x \nvDash C_D$), we first determine $\mathcal{D}_{\{IIS\}}$. It represents the set of all disjunct Irreducible Inconsistent Sets (IISs), representing subsets of clusters causing the schedule integration to be infeasible. Secondly, we apply an SMT approach to resolve all infeasibilities of $\mathcal{D}_{\{IIS\}}$ through adapting individual process start times. Finally, the adapted clusters are merged into a single cluster ($\underset{merge}{\forall D \in \mathcal{D}_{\{IIS\}}}$) to guarantee termination.

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 adapted 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 C_D is removed from $\mathcal{D}$. To determine all IISs, we first exploit the domain knowledge we obtained during the interval calculation. If the calculated interval $I_{D,\tilde{D}}$ is empty, no feasible solution exists. Hence, in a first step, we initialize $\mathcal{D}_{\{IIS\}}$ with all pairs of clusters with an empty interval:

$$\mathcal{D}_{\{IIS\}} = \{\{D, \tilde{D}\} | D, \tilde{D} \in \mathcal{D}, I_{D,\tilde{D}} = \{\}\} \tag{8}$$

After all initial pair-wise infeasibilities have been determined, we apply an extended deletion filter. While a common deletion filter only determines a single infeasibility, here we apply an iterative approach until the remaining set of clusters is feasible.

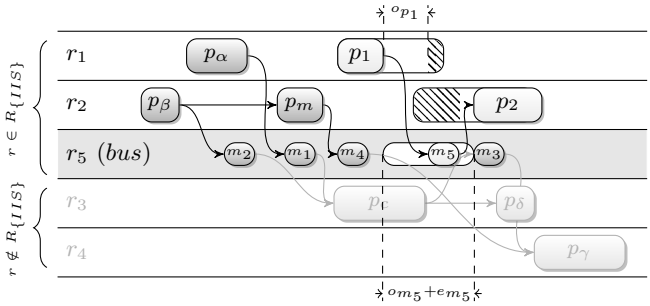


Figure 5: During conflict refinement we adapt the process offsets individually. This is only illustrated for one of the two applications here. Offset optimization is done concurrently, ensuring that precedence constraints are not violated (▨). Our algorithm only adapts the start times of processes mapped to resources used by multiple cluster in the unsatisfiable subsets ($r \in R_{\{IIS\}}$).

Algorithm 1 describes an Iterative Deletion Filter extending the IIS $\mathcal{D}_{IIS}$ until the remaining subset $\hat{\mathcal{D}}$ is feasible.

```
1: D̂ = D \ D_{IIS}
2: while ∀x : x ⊭ C_D̂ do
3:     for D ∈ D̂ do
4:         if ∀x : x ⊭ C_D̂ \ C_D then
5:             D̂ = D̂ \ D
6:         end if
7:     end for
8:     D_{IIS} = D̂ ∪ D_{IIS}
9:     D̂ = D \ D_{IIS}
10: end while
```

Algorithm 1 iteratively applies a deletion filter to the problem, until the subset $\hat{\mathcal{D}} \subset \mathcal{D}$ 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 removed from the infeasible set (line 5), otherwise it remains part of it. After an IIS has been determined, it is then added to $\mathcal{D}_{\{IIS\}}$ (line 8), and the process is repeated for an updated $\hat{\mathcal{D}}$ (line 9).

Resolve all IISs. To cope with infeasible problems, we propose an SMT formulation to adapt the start times of individual processes in addition to the cluster offset. To ensure all constraints defined for the cluster schedule are not affected by adapting process start-times, our approach ensures that the previously defined end-to-end delay is not exceeded. The SMT formulation is based on the following constraints:

- $s_p \in \mathbb{R}$ - start time for each process $p \in P$ as defined by the subsystem schedule.
- $f_p = s_p + e_p$ - process finish time.
- $w_{p,\tilde{p}} \in \mathbb{R}$ - waiting time between two data-dependent processes $p, \tilde{p}$. It is defined as the delay between the finish time of p and start of $\tilde{p}$.
- $H_r = \underset{p \in P_r}{lcm}(h_p)$ - is the hyper period of all processes $p \in P_r$ mapped to the resource r, where lcm defines the least common multiple. It defines the period after which the schedule for r repeats.
- F - set of processes of an application. $\mathcal{F}$ defines the set of applications F.
- $\pi = (p, \tilde{p})$ - defines the precedence for $p, \tilde{p} \in F$, defining the execution order. Π_F denotes the set of all π for application F.

Based on these constants our algorithm then determines a feasible schedule for the following variables:

- $\mathbf{o_D}$: cluster offset for cluster D.
- $\mathbf{o_p}$: process offset for process p.

Based on the precedence constraints defined in Π_F, $\mathbf{o_p}$ is limited by the finish time of directly preceding processes $\check{p}$ and the starting time of all directly succeeding processes $\hat{p}$; $(\check{p},p)(p,\hat{p}) \in \Pi_F$. p might then be shifted up to the finish time $f_{\check{p}}$ of its closest predecessor and the start time of its closest successor $s_{\hat{p}}$. Hence, the interval for $\mathbf{o_p}$ is defined by the waiting time after the closest predecessor $w_{\check{p},p} = s_p - f_{\check{p}}$ and to its closest successor $w_{p,\hat{p}} = s_{\hat{p}} - f_p$:

$$\mathbf{o_p} \in [-w_{\check{p},p}, w_{p,\hat{p}}] \tag{9}$$

If p is the source or the sink process in a task graph, we assume that it might only be shifted in one direction, to decrease the end-to-end delay but not to increase it. This ensures that all previously defined deadlines are not affected. Figure 5 illustrates the offsets for the three processes m_5, p_1 and p_2. (1) For process m_5, the predecessor $\check{m}_5 = p_1$ and the successor $\hat{m}_5 = p_2$ define the boundaries for $\mathbf{o_{m_5}}$ to $[-w_{p_1,m_5}+\mathbf{o_{p_1}}, w_{m_5,p_2}+\mathbf{o_{p_2}}]$. (2) To ensure all end-to-end delays are not affected by this process, the source process p_1 can only be shifted in one direction, reducing but not increasing the end-to-end delay ($\mathbf{o_{p_1}} \in [0, w_{p_1,m_5} + \mathbf{o_{m_5}}]$). (3) Similarly, for the sink process p_2: $\mathbf{o_{p_2}} \in [-w_{m_5,p_2} + \mathbf{o_{m_5}}, 0]$. Adjusting the subsystem structure using $\mathbf{o_p}$ in addition to the cluster offset $\mathbf{o_D}$ commonly resolves the determined infeasibilities, without violating previously defined constraints for the subsystem schedule. To improve the scalability of our algorithm, we only introduce an offset $\mathbf{o_p}$ for processes on a shared resource $r \in R_{\{IIS\}}$. The set $R_{\{IIS\}}$ defines all resources which are used by more than one cluster of the IIS. Accordingly, the process offset of predecessor or successor processes are only considered if these processes are executed on a shared resource. The following equations calculate feasible values $\mathbf{o_p}$ for each process together with cluster offsets $\mathbf{o_D}$, allowing to integrate all clusters:

$$\forall D \in \mathcal{D}_{\{IIS\}}: \quad 0 \leq \mathbf{o_D} < h_D \tag{10}$$

$$\forall r \in R_{\{IIS\}}, D \in \mathcal{D}_{\{IIS\}}, p \in D_r:$$

$$\mathbf{o_p} \geq \begin{cases} 0 & \text{if} \quad \nexists \check{p} : (\check{p},p) \in F \\ -w_{\check{p},p} + \mathbf{o_{\check{p}}} & \text{elseif} \quad \check{p} \in P_{\tilde{r}}, \tilde{r} \in R_{\{IIS\}} \\ -w_{\check{p},p} & \text{otherwise} \end{cases} \tag{11}$$

$$\mathbf{o_p} \leq \begin{cases} 0 & \text{if} \quad \nexists \hat{p} : (p,\hat{p}) \in F \\ w_{p,\hat{p}} - \mathbf{o_{\hat{p}}} & \text{elseif} \quad \hat{p} \in P_{\tilde{r}}, \tilde{r} \in R_{\{IIS\}} \\ w_{p,\hat{p}} & \text{otherwise} \end{cases} \tag{12}$$

$$\begin{aligned} &\forall r \in R_{\{IIS\}} : D, \tilde{D} \in \mathcal{D}_{\{IIS\}}, D \neq \tilde{D}, \forall p \in D_r, \forall \tilde{p} \in \tilde{D}_r, \\ &i = \{0, .., \tfrac{3 \cdot H_r}{h_p} - 1\}, j = \{0, .., \tfrac{3 \cdot H_r}{h_{\tilde{p}}} - 1\}, \\ &\mathbf{o_D} \in [0, h_D[, \mathbf{o_{\tilde{D}}} \in [0, h_{\tilde{D}}[: \\ &\quad \mathbf{o_D} + \mathbf{o_p} + i \cdot h_p + s_p + e_p \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o_{\tilde{p}}} + \mathbf{o_{\tilde{D}}} \\ &\oplus \quad \mathbf{o_D} + \mathbf{o_{\tilde{p}}} + j \cdot h_{\tilde{p}} + s_{\tilde{p}} + e_{\tilde{p}} \leq i \cdot h_p + s_p + \mathbf{o_p} + \mathbf{o_D} \end{aligned} \tag{13}$$

Constraint (10) first defines the range for the cluster offset. Constraints (11) and (12) then define the range for each process. We distinguish three cases: (a) The process is a source or a sink process and can only be shifted in right or left direction, respectively, reducing the end-to-end delay but not increasing it. (b) Predecessor or successor are part of $R_{\{IIS\}}$, hence the bound for $\mathbf{o_p}$ is variable and the offsets need to be optimized concurrently. (c) Predecessor or successor are no part of $R_{\{IIS\}}$, leading to a fixed bound. Finally, Constraint (13) ensures that two processes do not use a resource at the same time instant. It concurrently determines a cluster offset and a process offset. The SMT formulation assumes robust applications which benefit from a decreasing end-to-end delay. However, as control functions might be designed for a specific end-to-end delay, for these applications Constraints (11) and (12) might be adapted such that the position of the source and the sink processes within the cluster are fixed and only intermediate processes can be shifted. After a valid configuration was found, all clusters $D \in \mathcal{D}_{\{IIS\}}$ are merged to a single cluster D to prevent repeated optimization and ensure termination of the algorithm.

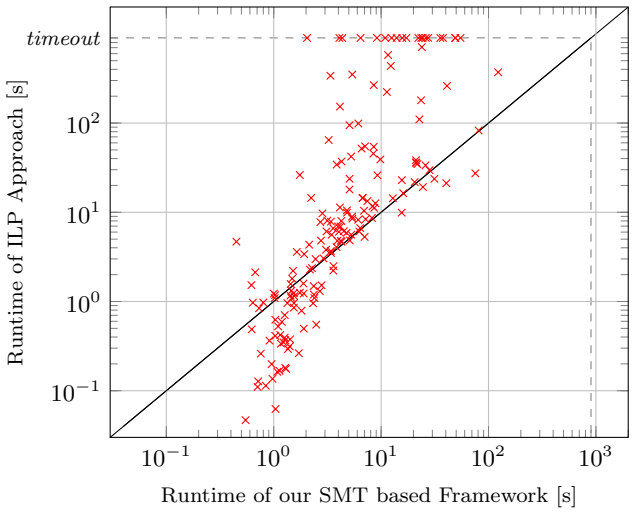


Figure 6: Runtime comparison of our schedule integration framework with existing ILP approach. For midsize and large problems our framework is significantly faster than the existing approach. The timeout was set to 15 minutes.

5. EXPERIMENTAL RESULTS

This section presents the results, using test cases and a case study to evaluate the effectiveness of the schedule integration module of our framework. First, we present a scalability analysis to show the benefits of our approach. Secondly, we present a realistic case study with a special focus on the conflict refinement. All calculations have been carried out on an Intel Xeon 3.2 GHz Quad Core with 12GB RAM. We use Microsoft's Z3 version 4.3.0 [15] as SMT solver. Note that the schedule is obtained at design time such that runtimes of several minutes are still acceptable.

5.1 Scheduling Result

We first compare our framework to an ILP approach [11]. Here, we apply an SMT-based scheduling derived from [11] to each application independently to generate independent cluster schedules. We then combine the cluster schedules by our schedule integration approach, including conflict refinement. The runtime of our framework is then compared to the ILP approach being applied to the same problem. As ILP solver we use CPLEX in version 12.2.

Our scalability analysis is based on 180 synthetic test cases consisting of up to 16 ECUs connected by an Ethernet bus, using up to 25 applications, with up to 100 tasks and 50 messages. Figure 6 illustrates the results of the runtime analysis. The results show that the ILP approach might have a slight runtime advantage of a few seconds for small and simple problems which is due to the overhead introduced through the iterative approach. However, with increasing size and complexity, our framework clearly outperforms the ILP approach. For 11.5% of the test cases, the ILP approach is unable to determine a solution within 15 minutes while our framework finds always feasible solutions in less than $100s$.

5.2 Framework Case Study

In the following, we present a realistic case study for our framework. The case study is based on a state-of-the-art au-

Table 1: Details of realistic case study

domain	applications	tasks	messages	ECUs[1]	application properties		
					periods	deadline	total execution[2]
body	3	15	10	7	$10.0 - 20.0ms$	$10.0 - 15.0ms$	$5.9 - 12.1ms$
chassis	4	26	11	8	$5.0 - 10.0ms$	$5.0 - 10.0ms$	$2.5 - 11.4ms$
information	3	13	10	5	$20.0ms$	$11.0 - 20.0ms$	$7.9 - 13.9ms$
electric	4	17	9	7	$5.0 - 20.0ms$	$5.0 - 12.0ms$	$0.8 - 8.4ms$
safety	6	27	16	10	$5.0 - 20.0ms$	$4.0 - 10.0ms$	$1.4 - 5.9ms$
telematics	1	6	6	6	$20.0ms$	$13.0ms$	$10.1ms$

[1] ECUs might be shared between different domains.
[2] Sum of execution time for all tasks and messages in application task graph. Might exceed deadline for parallel paths.

Table 2: Conflict refinement

iteration	number of clusters in IIS
1	2
2	2^3
3	5^3
4	2^3
5	2^3

[3] Includes merged cluster from previous IIS.

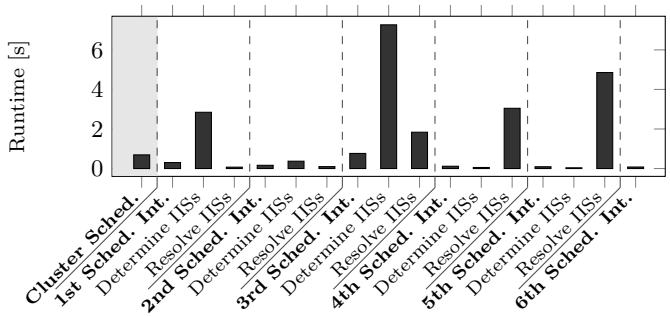


Figure 7: Runtime of the iterative steps during our consecutive schedule generation. After the independent cluster schedules have been created, our framework requires 6 iterations to generate a valid schedule.

tomotive architecture consisting of 32 ECUs connected by an Ethernet bus. Its functionality is implemented in 21 applications consisting of 104 tasks and 62 messages. The application attributes are outlined in Table 1.

Our framework computes a valid configuration within $22.68s$ while the ILP approach is unable to determine a solution within $24h$. Figure 7 illustrates the runtime contribution of each step of our framework, leading to the overall runtime. The results show that both the generation of independent cluster schedules as well as the iterative schedule integration require less than a second each. The conflict refinement (Determine IISs and Resolve IISs) in contrast, accounts for more than 90% of the runtime. During conflict refinement, clusters with conflicting schedules (IISs) are adjusted and merged to a single schedule. Hence, with each iteration the number of clusters decreases while the former IISs form a larger and more complex single cluster. *Determining all IIS* is an iterative process and depends on both the runtime of the schedule integration and the number of conflicting cluster sets in the IIS. Table 2 illustrates the number of clusters in the IISs for each iteration. It shows that the runtime increase during the third iteration is due to the larger set size. At the same time, the schedule integration has an increased runtime compared to the previous iterations, further increasing the runtime for determining all IISs. Similarly, the runtime for *resolving all IIS* increases with each iteration step. This is due to the rising problem complexity as all in the previous step resolved IISs are part of the consecutive one. Hence, the number of processes to consider increases with each step. However, as the runtime only increases gradually, the proposed schedule integration approach is well suited for scheduling large problems. Furthermore, it shows the applicability of our schedule integration approach as basis for a highly composable architecture in combination with a data-centric description.

6. CONCLUSION

This paper proposes a framework for modular time-triggered systems. It is based on a data-centric design approach leading to highly flexible and composable architectures. We present a framework for integrating independent software components or subsystems into a global system. It is based on a modular scheduling approach capable of integrating subsystem schedules independently. We first present an SMT approach to generate schedules for independent subsystems. Secondly, we propose a schedule integration approach which combines independent subsystem schedules into a global scheduling. Finally, a conflict refinement is presented which adapts individual subsystem schedules if no feasible solution can be found. Our test cases and case study give evidence of the clear runtime advantage of our approach compared to existing schedule optimization techniques. Furthermore, our schedule integration approach allows to incrementally add or update subsystem schedules, enabling highly modular architectures. In future work, we will extend our framework and address problems like task and message distribution during component integration.

7. REFERENCES

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

[2] AUTOSAR. AUTOSAR 4.1, 2013.

[3] I. Shin and I. Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. in Embedded Computing Systems*, 7(3):30:1–30:39, 2008.

[4] R.I. Davis and A. Burns. Hierarchical fixed priority pre-emptive scheduling. In *Proc. of RTSS*, pages 389–398, 2005.

[5] H. Kopetz and G. Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, 2003.

[6] 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 WISES*, pages 95–103, 2009.

[7] 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 DAC*, pages 874–877, 2009.

[8] H.T. Lim, L. Völker, and D. Herrscher. Challenges in a future IP/ Ethernet-based in-car network for real-time applications. In *Proc. of DAC 2011*, pages 7–12, 2011.

[9] 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.

[10] W. Steiner. An Evaluation of SMT-Based Schedule Synthesis for Time-Triggered Multi-hop Networks. In *Proc. of RTSS*, pages 375–384, 2010.

[11] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty. Modular Scheduling of Distributed Heterogeneous Time-Triggered Automotive Systems. In *Proc. of ASPDAC*, pages 665–670, 2012.

[12] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty. Schedule Integration for Time-Triggered Systems. In *Proc. of ASPDAC*, pages 52–58, 2013.

[13] D. Goswami, M. Lukasiewycz, R. Schneider, and S. Chakraborty. Time-triggered implementations of mixed-criticality automotive software. In *Proc. of DATE*, pages 1227–1232, 2012.

[14] Z. Wei, C. Jike, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *Proc. of ACSD*, pages 132–141, 2005.

[15] 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.