Multi-Schedule Synthesis for Variant Management in Automotive Time-Triggered Systems

Florian Sagstetter, Peter Waszecki, Sebastian Steinhorst,
Martin Lukasiewycz, Samarjit Chakraborty

***Abstract*—Car manufacturers provide a growing variety of models and configuration options for customers. In the highly competitive and cost-driven automotive industry, managing these variants and increasing the reuse of functionality in different variants has therefore become one of the key challenges. This paper addresses the problem of generating variant schedules for time-triggered Electrical/Electronic (E/E)-architectures. 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 applications common to different variants. This makes these applications variant-independent, thus, reduces 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. Finally, to address the increased complexity, we also propose a divide-and-conquer approach to partition the problem, improving the scalability.**

***Index Terms*—Distributed systems, schedule synthesis, variant management, automotive**

I. Introduction

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 [1]. 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 [2], techniques for variant management of Electrical/Electronic (E/E)-architectures have not been systematically studied so far.

This paper addresses the problem of a variant-aware schedule synthesis. We assume an architecture executing safety critical applications according to a time-triggered execution scheme to provide predictability, while remaining resources might be used by less critical applications according to an event-triggered execution scheme. In the work at hand, we focus on time-triggered schedule synthesis for variant-management of safety-critical applications. Fig. 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 variants, 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. However, 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 Electronic Control Units (ECUs) is facilitated as the configuration has to be done only once [3].

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

Florian Sagstetter, Peter Waszecki, Sebastian Steinhorst and Martin Lukasiewycz are with TUM CREATE, Singapore (firstname.lastname@tum-create.edu.sg).

Samarjit Chakraborty is with TU Munich, Germany (samarjit@tum.de).

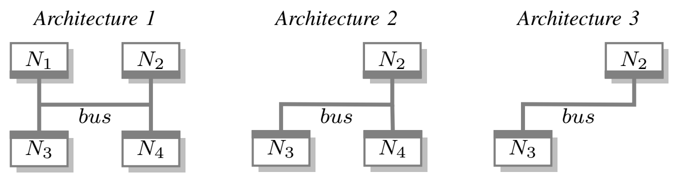
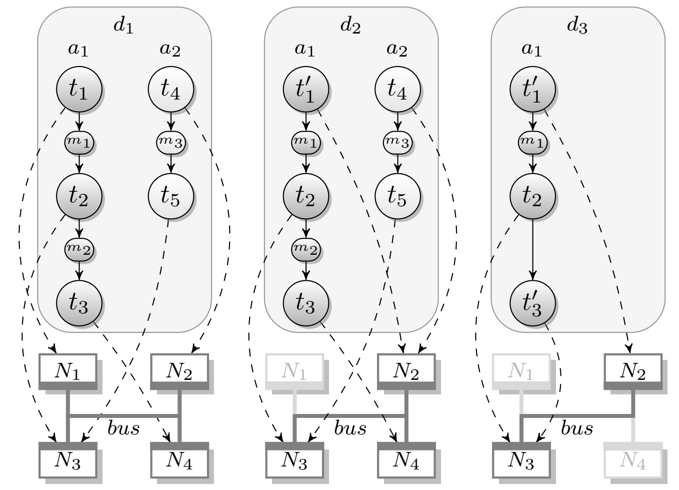
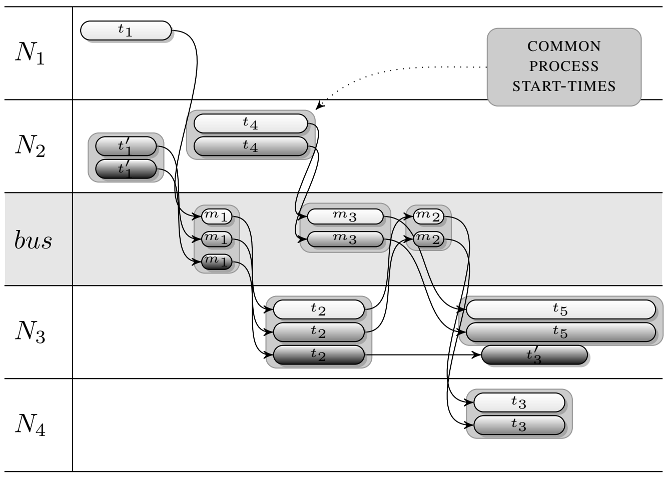


Fig. 1: Hardware architectures of three different variants, each consisting of a different number of Electronic Control Units (ECUs) connected over a bus.

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 [4], 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.



(a) Task graphs including a task to resource mapping for three variants



(b) Corresponding multi-schedule for the three variants d_1,d_2,d_3

Fig. 2: (a) Variant specification d_1 describes the task graph of the full system using 4 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. (b) A multi-schedule defines the same schedule for tasks and messages shared in multiple variants.

Methodology. Fig. 2a illustrates the task graphs of the three vehicle variants from Fig. 1, sharing common tasks and messages but differing in the underlying architecture. We use a direct acyclic task graph model where each vertex represents a task or message, and the edges represent their data-dependencies. We propose a schedule synthesis approach generating a time-triggered multi-schedule that defines release-times for periodic tasks and messages, as illustrated in Fig. 2b. 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 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., [5][6], 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., [7][8]. 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 [9][10][11], 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 [12] a graph-based representation to enable the use of graph theoretic analysis tools is presented, while [13] proposes a multi-variant based Design Space Exploration (DSE). Finally, a variant-aware schedule synthesis was first addressed in [3]. 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 V.

Contributions of the paper. This paper 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 a Satisfiability Modulo Theories (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 safety applications. The focus here lies on non-preemptive time-triggered scheduling as we exploit the temporal composability of time-triggered architectures [14][15] for the incremental extension and the partitioning. Variant-aware event-triggered scheduling requires a significantly different approach, and is part of future work. In addition, the automotive industry is moving towards time-triggered scheduling for predictability of safety-critical hard real-time functions. The focus here is not to obtain globally optimal schedules, but rather to determine variant schedules with minimal differences, satisfying predefined maximal 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.

Paper outline. The paper is organized as follows. Section II introduces our framework. Section III presents implementation details of our multi-schedule synthesis. Section IV evaluates our framework with three case studies and an automotive lab setup. Finally, Section V discusses related work, before Section VI concludes the paper.

II. Framework

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

A. Problem description

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 delays are defined for distributed applications [16]. Classical scheduling strategies such as Earliest Deadline First (EDF) or Rate-Monotonic (RM) are designed for single processor applications, and are therefore unsuitable candidates for scheduling applications with a high degree of distribution. Instead, we assume an architecture which provides predictability for safety-critical applications, while remaining resources might be used by less critical applications following an event-triggered execution scheme. Here, we focus on time-triggered scheduling for safety-critical tasks, defining release-times for periodic tasks. We consider a direct acyclic graph model where each vertex represents a task or message and the edges represent the dependencies between them.

Time-triggered scheduling. For 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 [17] or Automotive Ethernet based on the Time Sensitive Networking (TSN) standards [18] support time synchronization of ECUs, providing a global clock. A time-triggered schedule partitions the time and assigns the start-times for these partitions to periodic tasks and messages. For the sake of simplicity, in the following we refer to both task and message as process p. Each process is defined by its period h_p and execution time e_p. For a given start-time s_p, a process is executed during the time interval $\mathbf{t}$ given by:

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

We define an application as a set of processes connected by data-dependencies, e.g., variants d_1 and d_2 in Fig. 2a each define two applications a_1 and a_2. An application a is represented by a task graph $G_a = (P_a, E_a)$ consisting of the processes P_a with their data-dependencies E_a, as well as the maximal allowed end-to-end delay from source to sink tasks. While the periods of applications might differ, all processes of one application have the same period, due to data-dependencies. At design time, schedule synthesis defines a start-time s_p for each process while taking constraints like the maximum end-to-end delay of each application into account.

Multi-schedule synthesis problem. Here, we address multi-schedule synthesis for time-triggered systems which determines individual schedules for a set of variants. Fig. 2b 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 not to determine an optimal schedule, but rather to minimize the differences between variant schedules, 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.

B. 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.

Fig. 3 illustrates the first two iterations of the multi-schedule synthesis for the three variants in Fig. 2a. We first determine shared processes for all variants as illustrated in Fig. 3a. 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 Fig. 3b 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 Fig. 3c. 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 Fig. 3d, only taking processes into account which have not been scheduled yet. After a comprehensive task graph was created (Fig. 3e), we extend our multi-schedule as illustrated in Fig. 3f. 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, e.g., body, chassis, and safety, 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 overcome the high complexity and limited scalability of time-triggered scheduling [19][20]. 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 [21][22]. We therefore extend these *schedule integration* approaches, integrating separately generated partition schedules into a system schedule, with a partitioning approach 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

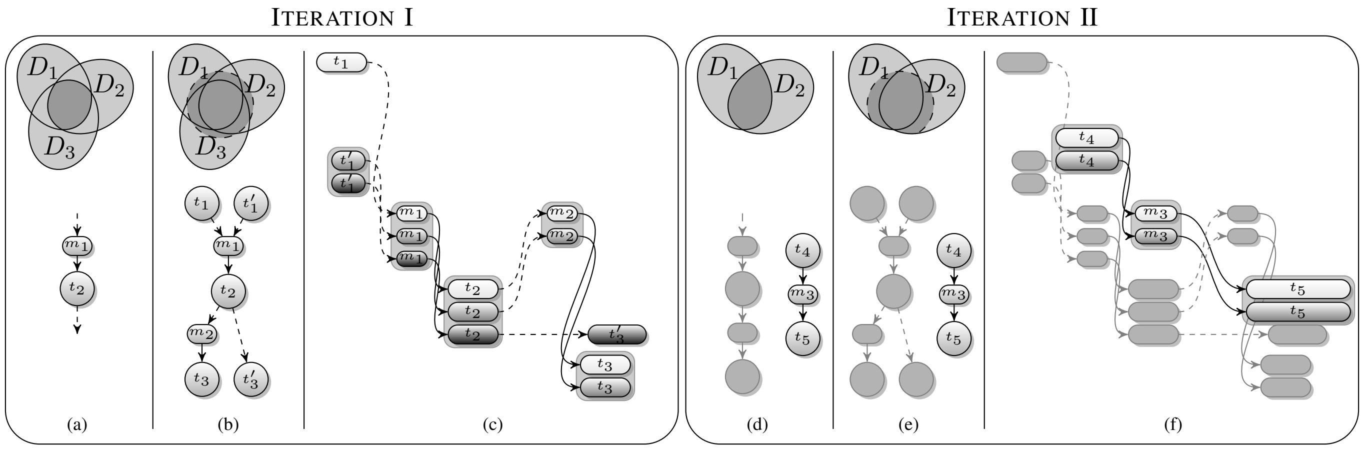
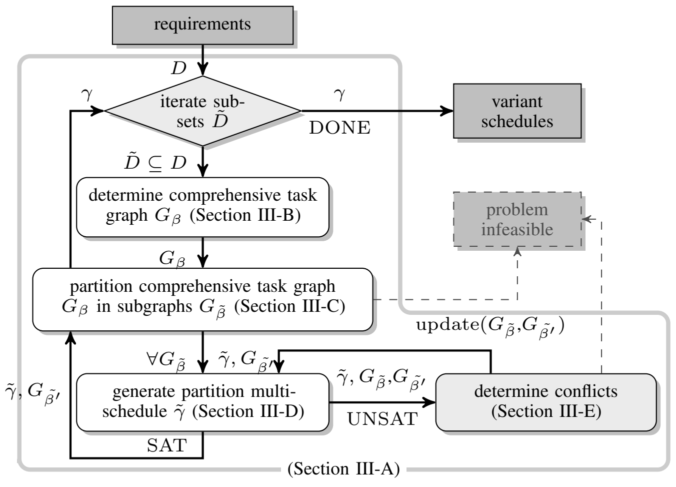


Fig. 3: First two iterations of a multi-schedule synthesis for variants from Fig. 2a. Each iteration (1) determines shared processes (a,d), (2) extends these to comprehensive task graphs (b,e), (3) generates a multi-schedule for the comprehensive task graph (c,f). Grayed out elements represent parts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



requirements
D
γ
iterate subsets $\tilde{D}$
γ
DONE
variant schedules
$\tilde{D} \subseteq D$
determine comprehensive task graph G_β (Section III-B)
problem infeasible
G_β
partition comprehensive task graph G_β in subgraphs $G_{\tilde{\beta}}$ (Section III-C)
update($G_{\tilde{\beta}}$,$G_{\tilde{\beta}'}$)
$\forall G_{\tilde{\beta}}$
$\tilde{\gamma}, G_{\tilde{\beta}'}$
$\tilde{\gamma}, G_{\tilde{\beta}'}$
$\tilde{\gamma}, G_{\tilde{\beta}}, G_{\tilde{\beta}'}$
generate partition multi-schedule $\tilde{\gamma}$ (Section III-D)
determine conflicts (Section III-E)
UNSAT
SAT
(Section III-A)

Fig. 4: 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.

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. 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, a schedule integration combines the subschedules in each iteration. Once all processes have been scheduled, the multi-schedule is converted into individual variant schedules. Fig. 4 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 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 provides detailed information about conflicting parameters to the system designer 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.

III. Multi-Schedule Synthesis

This section describes the multi-schedule framework, as illustrated in Fig. 4. 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 II-B. 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.

A. 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

Algorithm 1: Outer-loop of multi-schedule synthesis

```
   Input: set of variant specifications $d \in D$
   Output: multi-schedule $\gamma$
   // initialize multi-schedule $\gamma$ and comprehensive
      task graph for scheduled applications $G_{\beta'}$:
1  $\gamma = \emptyset$
2  $G_{\beta'} = (\emptyset, \emptyset)$
   // iterate through all variant subsets $\tilde{D}$:
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 $G_\beta$:
5          $\tilde{D} = \{\tilde{d} := G_{\tilde{d}} = (P_d \setminus P_{\beta'}, E_d \setminus E_{\beta'}) | d \in \tilde{D}\}$
6          $G_\beta(P_\beta, E_\beta) =$
           $\mathrm{getComprehensiveTaskGraph}(\tilde{D})$
7          if $P_\beta \neq \emptyset$ then
               // determine multi-schedule $\gamma$ and
                  update $G_{\beta'}$:
8              $\gamma = \mathrm{generateMultiSchedule}(\beta, \beta', \gamma)$
9              $G_{\beta'} = (P_{\beta'} \cup P_\beta, E_{\beta'} \cup E_\beta)$
10         end
11     end
12 end
```

Algorithm 2: Determine comprehensive task graphs

```
1  Function getComprehensiveTaskGraph ($(\tilde{D})$)
       Input: subset of variant specifications $\tilde{D} \subseteq D$
       Output: comprehensive task graph $G_\beta$
       // determine processes $\tilde{P}$ common to all
          variants in $\tilde{D}$:
2      $\tilde{P} = \bigcap_{\tilde{d} \in \tilde{D}} P_{\tilde{d}}$
3      if $\tilde{P} \neq \emptyset$ then
           // initialize comprehensive task graph $G_\beta$:
4          $G_\beta = (\emptyset, \emptyset)$
           // iterate through all applications $a$ of
              each variant $\tilde{d}$:
5          for $\tilde{d} \in \tilde{D}$ do
6              for $a \in A_{\tilde{d}}$ do
                   // if $a$ contains a process common to
                      all variants in $\tilde{D}$, add $a$ to $G_\beta$:
7                  if $P_a \cap \tilde{P} \neq \emptyset$ then
8                      $G_\beta = (P_\beta \cup P_a, E_\beta \cup E_a)$
9                  end
10             end
11         end
12         return $G_\beta$
13     else
14         return $(\emptyset, \emptyset)$
15     end
16 end
```

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 the schedule synthesis might become intractable, efficient solvers such as Z3 [23] 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.
- G_β - a comprehensive task graph $G_\beta = (P_\beta, E_\beta)$, describing the relations of different variants to shared tasks, as illustrated in Fig. 3b.
- $\gamma(p) : P \to \mathbb{R}$ - multi-schedule defining the start-times of processes. It returns the process start-time s_p for a process $p \in P$, see Fig. 3c.

Algorithm 1 outlines the outer-loop for our multi-schedule synthesis. The algorithm first initializes an empty comprehensive task graph $G_{\beta'}$ and an empty schedule γ (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_β 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 γ and $G_{\beta'}$ are updated (lines 8-9). Fig. 3 illustrates two iterations of this algorithm.

The functions $\mathrm{getComprehensiveTaskGraph}(\tilde{D})$, determining a comprehensive task graph, and $\mathrm{generateMultiSchedule}(\beta, \beta', \gamma)$, determining a multi-schedule, are described in detail in Sections III-B and III-C till III-E, respectively. Note that if none of the variants shares any commonality with another variant, an individual schedule is created for each variant.

B. 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.

To determine common subgraphs, usually the maximum common subgraph-isomorphism problem has to be solved which is known to be NP-hard [24]. 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 2 determines the comprehensive task graphs 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 Fig. 3a). If a common induced subgraph exists (line 3), we initialize an empty task graph G_β (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_β (line 8). Finally, the algorithm returns G_β (line 12), or, if no common subset exists, it returns an empty task graph (line 14). Fig. 3b shows the comprehensive task graph G_β 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.

C. 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. Fig. 5a 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 N_4. Hence, the problem might be partitioned following this domain-based architecture. Partitioning of the problem to reduce the search space has proven beneficial to clearly improve the runtime of solving time-triggered scheduling problems [21][22].

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 $G_\beta = (P_\beta, E_\beta)$ to a representation $G_{C_\beta} = (A_{C_\beta}, E_{C_\beta})$ with applications A_{C_β} as vertices and their resource dependencies as edges. Fig. 5b illustrates such a graph for the applications illustrated in Fig. 5a. For instance, application a_2 and a_3 execute one task each on ECU N_4 and are therefore connected by one edge. Similarly, a_1 and a_2 share three resources, N_2, N_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 $c_1 = \{a_1, a_2\}$ and $c_2 = \{a_3, a_4\}$ separately, and the partition schedules are integrated in a second step.
The partitioning is defined as:

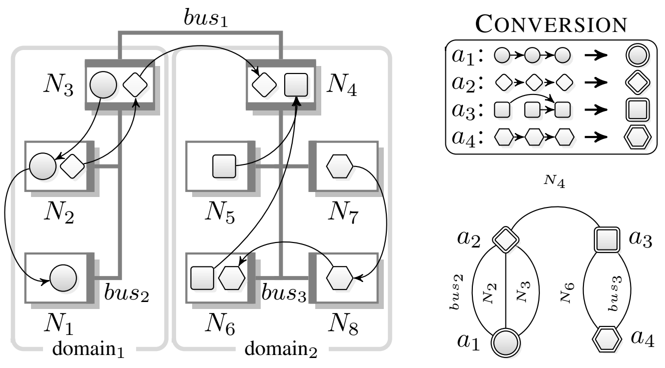
$$B = \text{partition}(\beta)$$

$$\tilde{\beta} \in B,\ G_{\tilde{\beta}} = (P_{\tilde{\beta}}, E_{\tilde{\beta}}):\ P_{\tilde{\beta}} \subseteq P_\beta, E_{\tilde{\beta}} \subseteq E_\beta$$

Additionally we introduce the following parameter.

- $c \in C_\beta$ - cluster representing a subset of applications of variant specification G_β as Graph $G_c = (A_c, E_c)$.

Metrics. The partitioning metric presented in the following is not only limited to domains, but also determines 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:



(a) Domain based architecture (b) Application relations

Fig. 5: (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.

$$m_{\text{imbalance}}(C_\beta) = \underbrace{\frac{1}{|C_\beta|} \cdot \sum_{c \in C_\beta}}_{\text{average over all clusters}} \sqrt{\left(\underbrace{\frac{|A_c|}{\frac{|A_{C_\beta}|}{|C_\beta|}}}_{\text{deviation of } |A_c| \text{ from average node number per cluster}} - 1 \right)^2} \tag{1}$$

With this metric we want to determine how much the number of nodes per cluster deviates from a perfectly balanced partitioning where each c would obtain the same $|A_c|$. 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.

$$\begin{aligned} &\text{maxcross}(c) = \\ &\underset{|e|}{\text{argmax}} \left\{ |e| \,\middle|\, a_i \in A_c \wedge a_j \in A_{\tilde{c}} \right\} \text{ for all } \tilde{c} \in C_\beta \setminus c \end{aligned} \tag{2}$$

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.

$$m_{\text{avgcross}}(C_\beta) = \frac{1}{|C_\beta|} \sum_{c \in C_\beta} \text{maxcross}(c) \tag{3}$$

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 $m_{\text{imbalance}}(C_\beta)$ is achieved while the subgraphs have an acceptable number of interlacing edges $m_{\text{avgcross}}(C_\beta)$:

$$\text{minimize } m_{\text{imbalance}}(C_\beta) \text{ s.t. } m_{\text{avgcross}}(C_\beta) < \epsilon_{\text{cross}} \tag{4}$$

For this purpose, we apply the efficient polynomial time Girvan-Newman algorithm [25] 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 $m_{\text{imbalance}}(C_\beta)$ and $m_{\text{avgcross}}(C_\beta)$ and decide whether the result of the graph partitioning is suitable according to our metrics. Consequently, we only start to evaluate $m_{\text{imbalance}}(C_\beta)$ once the first splitting in the graph has occurred, as an unpartitioned graph is inherently balanced. We only accept a partition if $m_{\text{avgcross}}(C_\beta)$ is below a threshold of ϵ_{cross}. As $m_{\text{avgcross}}(C_\beta)$ 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 $m_{\text{imbalance}}(C_\beta)$ 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 $m_{\text{imbalance}}(C_\beta)$ 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 III-D for each subgraph individually. To re-integrate the individual partition schedules into a global schedule, we apply a schedule integration approach. During schedule integration a temporal offset $\mathbf{o}_\beta$ is defined for each partition schedule. This offset maintains the general structure of the partition schedule and does not affect the subsystem behavior, as the start times of all processes within the partition are adapted concurrently and the partition schedule is executed periodically. To determine whether p utilizes a resource for a specific point in time $\mathbf{t}$, we define the following function:

$$\eta(p,\mathbf{t}) = \begin{cases} 1 & \forall \mathbf{t} : s_p + n \cdot h_p \le \mathbf{t} \le s_p + n \cdot h_p + e_p, n \in \mathbb{N}_0 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Here, s_p is the start-time, e_p the execution time, and h_p the period of process p. The two partitions β and $\tilde{\beta}$ can then be integrated in a global multi-schedule, if the following equation holds:

$\forall p \in P_\beta, \forall \tilde{p} \in P_{\tilde{\beta}}, r(p) = r(\tilde{p}), \mathbf{t} \in \mathbb{R}^+$:

$$\eta(p,\mathbf{t}-\mathbf{o}_\beta) + \eta(\tilde{p},\mathbf{t}-\mathbf{o}_{\tilde{\beta}}) \le 1 \tag{6}$$

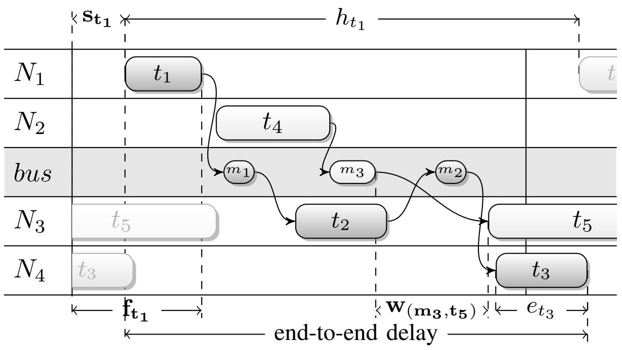


Fig. 6: Time-triggered schedule for variant specification d_1. The schedule defines a start-time s_p for each periodically executed task or message. Instead of considering task deadlines, we define a maximum end-to-end-delay for the application.

Here, the function $r(p)$ returns the resource a process is mapped to. Hence, $\mathbf{o}_\beta$ and $\mathbf{o}_{\tilde{\beta}}$ must be selected such that no two processes $p \in P_\beta, \tilde{p} \in P_{\tilde{\beta}}$ intersect when scheduled on the same resource. The partition schedule is executed with the period $h_\beta = \operatorname*{lcm}_{p \in P_\beta}(h_p)$, defining the least common multiple of all process periods in partition β. The offset $\mathbf{o}_\beta$ might therefore have any value between 0 and h_β. To determine offsets for each partition, we use a two step approach. (1) Feasible intervals for the relative offset of each partition pair $\beta, \tilde{\beta} \in B$ are computed. The relative offset is defined as $\mathbf{o}_\beta - \mathbf{o}_{\tilde{\beta}}$, hence it defines the relation between the two offsets. (2) The final offsets for the whole set are determined with an SMT solver. Based on Equation (6), the intervals defining all feasible offsets are computed as follows:

$$\begin{aligned} \Delta_{(\beta,\tilde{\beta})} = \{x | x = \mathbf{o}_\beta - \mathbf{o}_{\tilde{\beta}}, \mathbf{o}_\beta \in [0, h_\beta], \mathbf{o}_{\tilde{\beta}} \in [0, h_{\tilde{\beta}}], \\ p \in P_\beta, \tilde{p} \in P_{\tilde{\beta}}, r(p) = r(\tilde{p}), \\ \exists \mathbf{t} : \eta(p, \mathbf{t} - \mathbf{o}_\beta) + \eta(\tilde{p}, \mathbf{t} - \mathbf{o}_{\tilde{\beta}}) \le 1\} \end{aligned} \tag{7}$$

For each process pair p and $\tilde{p}$ being part of different partitions, but being mapped to the same resource, the feasible relative offsets are determined. $\Delta_{(\beta,\tilde{\beta})}$ represents the intersection of these offsets for all process pairs, containing all feasible intervals $\delta_{(\beta,\tilde{\beta})} = [\underline{\delta}_{(\beta,\tilde{\beta})}, \overline{\delta}_{(\beta,\tilde{\beta})}]$ for the relative partition offset for β, $\tilde{\beta}$. Based on these intervals, we apply the following SMT formulation to determine the partition offsets $\mathbf{o}_\beta$:

$\forall \beta \in B$:

$$0 \le \mathbf{o}_\beta < h_\beta \tag{8}$$

$\forall \beta, \tilde{\beta} \in B, \beta \neq \tilde{\beta}$:

$$\bigoplus_{\forall \delta_{(\beta,\tilde{\beta})} \in \Delta_{(\beta,\tilde{\beta})}} \underline{\delta}_{(\beta,\tilde{\beta})} \le \mathbf{o}_\beta - \mathbf{o}_{\tilde{\beta}} \le \overline{\delta}_{(\beta,\tilde{\beta})} \tag{9}$$

Here, the symbol $\oplus$ represents an *exclusive or* (XOR) operation. Constraint (8) defines the boundaries for $\mathbf{o}_\beta$, while Constraint (9) ensures that all determined $\mathbf{o}_\beta$ lie in the previously determined intervals. Updating the process start-times for each partition with the obtained offsets leads to the global multi-schedule.

D. 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 Fig. 3f. The multi-schedule represents all individual variant schedules. For instance, Fig. 6 shows the respective schedule for variant specification d_1.

We assume a bus that allows to freely allocate time slots, such as for the upcoming Automotive Ethernet based on TSN. This allows to define a schedule with flexible message release-times rather than packing messages into predefined slots as it is done for other time-triggered buses like FlexRay. Consequently, we define a message by its transmission time and treat both messages and tasks as processes. The schedule synthesis is based on the following additional parameters:

- p - process, referring to both tasks and messages.
- h_p - process period, time after which p is repeated.
- e_p - execution time of 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.
- $\delta(p) : P \rightarrow D$ - returns all variant specifications containing a process p.
- $H(p, \tilde{p}) = \operatorname{lcm}(h_p, h_{\tilde{p}}) : P \rightarrow \mathbb{R}$ - returns hyper period of p and $\tilde{p}$ where lcm defines the least common multiple. Defines period after which 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.
- $\Phi(E_a)$ - returns all paths ϕ of application a. Applies a *Depth-first search* algorithm to determine paths [26].

Based on these parameters our algorithm 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_β.
$\forall p \in P_\beta :$

$$0 \leq \mathbf{s_p} < h_p \tag{10}$$

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

$$\begin{aligned} &i \cdot h_p + \mathbf{s_p} + e_p \leq j \cdot h_{\tilde{p}} + \mathbf{s_{\tilde{p}}} \\ \oplus \quad &j \cdot h_{\tilde{p}} + \mathbf{s_{\tilde{p}}} + e_{\tilde{p}} \leq i \cdot h_p + \mathbf{s_p} \end{aligned} \tag{11}$$

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

$$\mathbf{f_p} = (\mathbf{s_p} + e_p)\% h_p \tag{12}$$

$$\mathbf{w_{(p,\tilde{p})}} = (\mathbf{s_{\tilde{p}}} - \mathbf{f_p})\% h_p \tag{13}$$

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

$$\theta_a \geq \sum_{p \in \phi} e_p + \sum_{(p,\tilde{p}) \in \phi} \mathbf{w_{(p,\tilde{p})}} \tag{14}$$

Constraint (10) defines the boundaries for the start time. As processes are periodically executed, their start-time cannot exceed their process period h_p. Constraint (11) 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. Constraint (12) calculates the finish time $\mathbf{f_p}$. Constraint (13) determines the waiting time between two data-dependent processes $p, \tilde{p}$. Finally, Constraint (14) 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. The modulo operation $\%$ in Constraints (12) and (13) is defined as a function in our SMT formulation with the following properties:

$$0 \leq a\%m < m \tag{15}$$

Hence, for $a < 0$ it returns a positive value, e.g., for $-m < a < 0 : a + m$.

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 $P_{\beta'}$ scheduled in previous iterations are considered as temporal intervals where processes of the current comprehensive task graph P_β cannot be scheduled if they are part of the same variant and are mapped to the same resource, see grayed out parts in Fig. 3f. 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 (11), we define the following constraints:
$\forall a \in A_{\beta'}$:

$$0 \leq \mathbf{o_a} < h_a \tag{16}$$

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

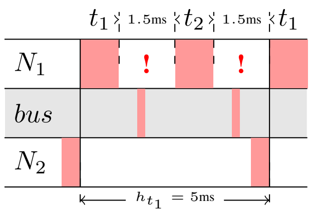
$$\begin{aligned} &i \cdot h_p + \mathbf{s_p} + e_p \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o_a} \\ \oplus \quad &j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o_a} + e_{\tilde{p}} \leq i \cdot h_p + \mathbf{s_p} \end{aligned} \tag{17}$$

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

$$\begin{aligned} &i \cdot h_p + s_p + \mathbf{o_a} + e_p \leq j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o_{\tilde{a}}} \\ \oplus \quad &j \cdot h_{\tilde{p}} + s_{\tilde{p}} + \mathbf{o_{\tilde{a}}} + e_{\tilde{p}} \leq i \cdot h_p + s_p + \mathbf{o_a} \end{aligned} \tag{18}$$

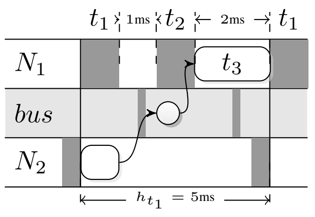
Constraint (16) defines the boundaries for the application offset. Constraint (17) ensures that a currently scheduled process does not intersect with already scheduled processes. The application offsets $\mathbf{o_a}$ hereby allow to alter the structure of parts which have already been scheduled in a previous iteration. Finally, Constraint (18) ensures that altering the application offsets does not lead to intersections between previously scheduled applications.

p	e_p	h_p	$r(p)$	s_p
t_1	1ms	5ms	N_1	0ms
t_2	1ms	5ms	N_1	2.5ms
t_3	2ms	10ms	N_1	!



(a) Conflicting schedule

p	e_p	h_p	$r(p)$	s_p
t_1	1ms	5ms	N_1	0ms
t_2	1ms	5ms	N_1	2ms
t_3	2ms	10ms	N_1	3ms



(b) Resolved conflict

Fig. 7: Exemplary illustration of a conflicting schedule and conflict refinement for non-preemptive scheduling. (a) Application a_1, containing t_1 and t_2, was scheduled in a previous iteration. However, the created schedule which is illustrated by the grayed out areas, conflicts with the execution time of t_3, part of the currently scheduled application a_2. (b) During conflict refinement the start-time of t_2 is adapted to support both applications. For the sake of simplicity, all other tasks and task dependencies are omitted for this example.

E. 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. Fig. 7 gives an example of how a conflict might be resolved. Our conflict refinement approach first determines conflicting application schedules and updates the comprehensive task graph G_β and the comprehensive task graph of previously scheduled processes $G_{\beta'}$ to resolve the conflicts, as illustrated in Fig. 4.

To determine conflicting application schedules in G_β and $G_{\beta'}$, we determine an Irreducible Inconsistent Set (IIS). An IIS represents a smallest set of conflicting constraints which might be resolved by removing any of these conflicting constraints. While modern SMT solvers already allow to determine an IIS, domain knowledge is not taken into account. Therefore, we apply a group-based approach determining IISs for groups of constraints, each group representing an application. We use a common deletion filter which 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_β might contain multiple IISs, we apply the deletion filter multiple times until all IISs have been determined. For more details please refer to [27].

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

IV. 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 Integer Linear Programming (ILP) approach, i.e., generating an individual schedule for each variant

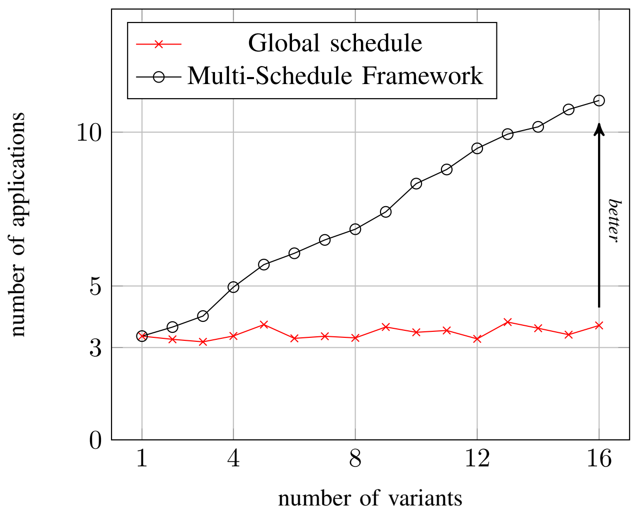


Fig. 8: Comparison of application number supported by a single global schedule for all variants, and a multi-schedule generating an individual schedule for each variant. A higher number of applications for a set of variants indicates a more efficient resource utilization. To improve legibility, the test cases are grouped by their variant number and the average value is displayed.

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 [23]. Note that the schedule is obtained at design time such that runtimes of several minutes are still acceptable.

A. 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 a non-variant-aware ILP approach 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 [28].

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. 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.

Fig. 8 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[1]. 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.

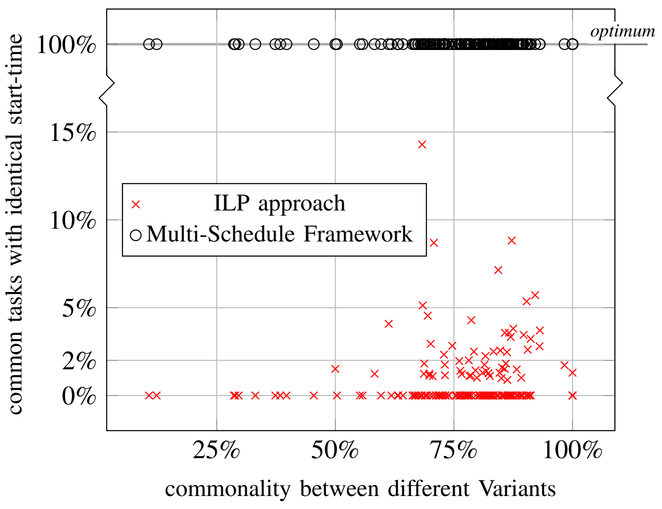


Fig. 9: Analysis of differences in schedules created by a non-variant-aware ILP approach and our variant-aware framework, depending on the ratio of common tasks to all tasks including variant specific tasks. We consider tasks to share the same schedule if their start-times differ less than 0.1ms.

B. 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.

To evaluate the approaches, we use 250 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. The number of processes common to the variants range from 10% to 100% of all tasks and messages. For the graph partitioning, we have determined a threshold of $\epsilon_{\text{cross}} = 1.3$ for our average crossing metric (see Section III-C) as beneficial through an experimental analysis. 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 analysis of both approaches.

Variant-awareness. Fig. 9 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 0.1ms. A detailed analysis has shown that the average difference for common parts lies between 2ms and 33ms 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 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

[1]As common and individual applications are selected randomly, for this case study the resulting average application number supported by a multi-schedule might be lower than the number of variants.

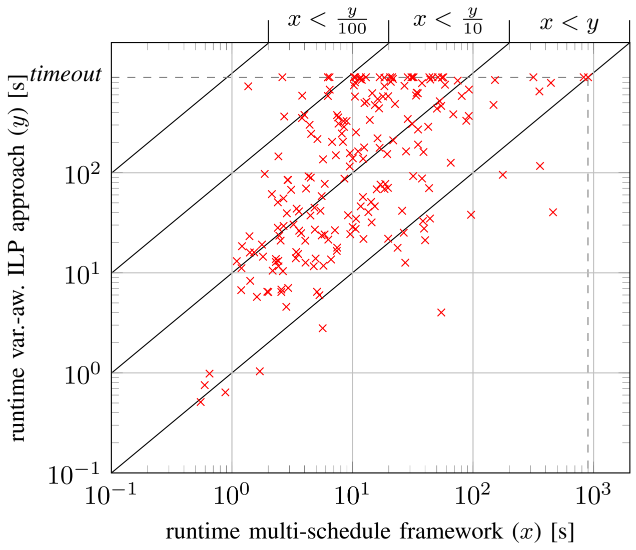


Fig. 10: Runtime comparison of our framework generating a multi-schedule with a variant-aware ILP approach generating all variant schedules concurrently.

long as their maximum end-to-end delay is not exceeded. These results are obtained without applying an objective function.

C. 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 250 synthetic test cases introduced in the previous section.

Figure 10 shows the result 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 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 III-C 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.

D. Automotive Case Study

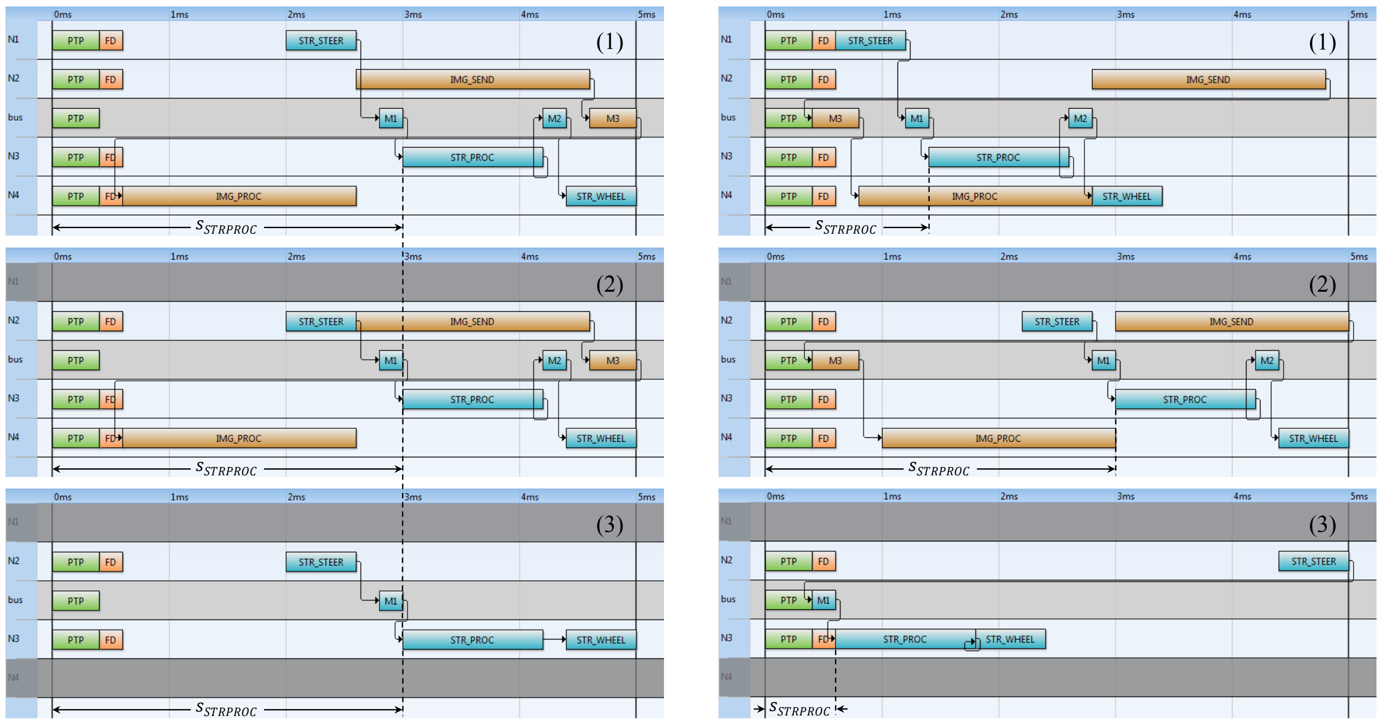
To illustrate the importance of a variant-aware schedule synthesis, in the following, we apply multi-schedule synthesis for the lab setup of a time-triggered automotive architecture. The system is able to adapt the hardware architecture and switch between predefined schedules at runtime. This allows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

We have defined 9 variants for our experimental prototype, of which 3 are shown in Fig. 2a. 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 the variant-awareness analysis in Section IV-B. Note that this case study gives an example where determining a single global schedule is not feasible, as it 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 50ms while the ILP takes 90ms. 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*. Fig. 11 illustrates the schedules generated by both methods for the variants illustrated previously in Fig. 2a. While the variant schedules generated with our framework (Fig. 11a) take the commonality of multiple variants into account, the schedules generated by the ILP (Fig. 11b), 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 Fig. 11b1-3, respectively. Fig. 11b 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 Fig. 11a, 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 Fig. 11b, 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 Fig. 11a. This can, in particular, be seen in Fig. 11a3 where a delay is introduced between



(a) Schedules for three variants created by our framework

(b) Schedules for three variants created by variant-unaware ILP

Fig. 11: (a) Variant schedules created by our framework. (b) Same variants determined by an ILP approach, generating variant-unaware schedules. The variant task graphs are defined in Fig. 2a. Note that we follow a periodic execution model. Hence, actuator tasks (e.g. t_{IMGPROC}) might appear to be executed before sensor tasks (e.g. t_{IMGSEND}), however, in this case data generated in the previous period is processed.

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.

V. Related 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 [9][10] and complex calibration processes are structured [11]. 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 [12] 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 [13]. 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. Nevertheless, the system models of functional variants might be used as input for our multi-schedule framework in order to generate suitable variant schedules.

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, [30] 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, [31] 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, various research has been carried out on modular systems [32][33]. It addresses the problem of integrating independent local schedulers into a global scheduling through assigning runtime budgets. This allows to apply different scheduling strategies such as EDF or RM for individual schedulers. While component-based scheduling is often limited to single resources, approaches like [34] circumvent this issue by splitting the global end-to-end delay into local deadlines which allows to assign a runtime budget to each single task. An extension of this approach would make component-based scheduling theoretically applicable for generating variant schedules with common properties. However, with a rising number of variants, the time-partitioning of end-to-end delays in local deadlines would strongly constrain the obtainable variant schedules, limiting the applicability of this approach.

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, an algorithm to determine an upper bound for mode change transition latencies for communication-based applications is presented in [5], targeting a static preemptive priority-based scheduling and asynchronous mode change protocols. By contrast, the authors in [6] use Time Division Multiple Access (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 [7] 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 [8] 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.

Finally, various work has been done in the area of time-triggered scheduling. For instance, in [38] an SMT-based approach to generate time-triggered schedules for *TTEthernet* is presented. Two ILP-based approaches for concurrent task and message scheduling are proposed for the automotive *FlexRay* bus in [19][20]. While [19] addresses the problem on *job-level*, [20] applies the schedule optimization on *task-level*. Finally, two approaches for schedule integration have been presented in [21] and [22]. Here, these approaches have been adapted for multi-schedule synthesis and extended by a partitioning. 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.

A first approach for a variant-aware schedule synthesis was presented in [3]. 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 [3], relying on a graph-based representation and an iterative SMT approach.

VI. Concluding Remarks

This paper addresses 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 propose 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 present 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, show the benefits of our approach. The resource requirements are clearly improved compared to a single global schedule, used in all variants, while the deviation between variants is clearly reduced compared to a variant-unaware approach, generating individual schedules without taking commonality into account. At the same time, testing and integration efforts are reduced compared to a variant-unaware approach, as they only have to be done once for common applications.

In future work, we will extend our framework 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 will then allow to determine variant schedules for optimized system performance while the development cost is minimized due to variant-awareness. Finally, an important line of future work is also to investigate variant-aware event-triggered scheduling.

References

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

[2] M. Lupa, “7 questions on MQB.” *Volkswagen Das Auto. Magazine*, Nov. 2012.

[3] J. Dvorak and Z. Hanzalek, “Multi-variant time constrained FlexRay static segment scheduling,” in *Proc. of WFCS*, May 2014, pp. 1–8.

[4] J. Sobotka and J. Novak, “Automation of automotive integration testing process,” in *Proc. of IDAACS*, Sep. 2013, pp. 349–352.

[5] 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 ETFA*, Sep. 2011, pp. 1–10.

[6] A. Azim, G. Carvajal, R. Pellizzoni, and S. Fischmeister, “Generation of communication schedules for multi-mode distributed real-time applications,” in *Proc. of DATE*, Mar. 2014, pp. 1–6.

[7] H. Zhao and R. Sakellariou, “Scheduling multiple DAGs onto heterogeneous systems,” in *Proc. of IPDPS*, Apr. 2006.

[8] M. Iverson and F. Ozguner, "Dynamic, competitive scheduling of multiple DAGs in a distributed heterogeneous environment," in *Proc. of HCW 98*, Mar. 1998, pp. 70–78.
[9] pure-systems GmbH, "Variant management with pure::variants," *Technical Article*, 2006.
[10] Mentor Graphics, "Top-down design of distributed embedded systems in light of timing considerations," *Technical Article*, 2010.
[11] Vector, "From pilot studies to production development," *Technical Article*, May 2008.
[12] T. Braun and F. Deubzer, "New variant management using multiple-domain mapping," in *Proc. of DSM*, Oct. 2007, pp. 363–372.
[13] S. Graf, M. Glaß, J. Teich, and C. Lauer, "Multi-variant-based design space exploration for automotive embedded systems," in *Proc. of DATE*, Mar. 2014, pp. 1–6.
[14] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proc. of the IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.
[15] 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*, Jun. 2009, pp. 95–103.
[16] 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 Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 7, pp. 979–992, Jul. 2009.
[17] B. Tanasa, U. Bordoloi, P. Eles, and Z. Peng, "Scheduling for fault-tolerant communication on the static segment of FlexRay," in *Proc. of RTSS*, Nov. 2010, pp. 385–394.
[18] IEEE, "Time-Sensitive Networking Task Group," 2015, http://www.ieee802.org/1/pages/tsn.html.
[19] 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*, Jul. 2009, pp. 874–877.
[20] M. Lukasiewycz, R. Schneider, D. Goswami, and S. Chakraborty, "Modular scheduling of distributed heterogeneous time-triggered automotive systems," in *Proc. of ASP-DAC*, Jan. 2012, pp. 665–670.
[21] F. Sagstetter, M. Lukasiewycz, and S. Chakraborty, "Schedule integration for time-triggered systems," in *Proc. of ASP-DAC*, Jan. 2013, pp. 52–58.
[22] 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 DAC*, Jun. 2014.
[23] L. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2008, vol. 4963, pp. 337–340.
[24] G. Levi, "A note on the derivation of maximal common subgraphs of two directed or undirected graphs," *CALCOLO*, vol. 9, no. 4, pp. 341–352, 1973.
[25] M. Girvan and M. E. Newman, "Community structure in social and biological networks," *Proc. of the National Academy of Sciences*, vol. 99, no. 12, pp. 7821–7826, Apr. 2002.
[26] S. Even, *Graph algorithms*. Cambridge University Press, 2011.
[27] J. W. Chinneck, *Feasibility and Infeasibility in Optimization:: Algorithms and Computational Methods*. Springer, 2007, vol. 118.
[28] ILOG, "CPLEX," http://www.ilog.com/products/cplex/.
[29] M. Dalkilic and V. Pitchumani, "A multi-schedule approach to high-level synthesis," in *Proc. of ICCD*, Oct. 1994, pp. 572–575.
[30] Q. Zhu, Y. Yang, M. Di Natale, E. Scholte, and A. Sangiovanni-Vincentelli, "Optimizing the software architecture for extensibility in hard real-time distributed systems," *IEEE Trans. on Ind. Informat.*, vol. 6, no. 4, pp. 621–636, Nov. 2010.
[31] P. Emberson and I. Bate, "Stressing search with scenarios for flexible solutions to real-time task allocation problems," *IEEE Trans. on Softw. Eng.*, vol. 36, no. 5, pp. 704–718, Sep. 2010.
[32] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. in Embedded Computing Systems*, vol. 7, no. 3, pp. 30:1–30:39, Apr. 2008.
[33] R. Davis and A. Burns, "Hierarchical fixed priority pre-emptive scheduling," in *Proc. of RTSS*, Dec. 2005, pp. 389–398.
[34] 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 SAC*, Mar. 2012, pp. 1821–1828.
[35] N. Stoimenov, S. Perathoner, and L. Thiele, "Reliable mode changes in real-time systems with fixed priority or EDF scheduling," in *Proc. of DATE*, Mar. 2009, pp. 99–104.
[36] V. Nelis, J. Goossens, and B. Andersson, "Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms," in *Proc. of ECRTS*, Jul. 2009, pp. 151–160.
[37] R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture," *IEEE Trans. on Comput.-Aided Design Integr. Circuits Syst*, vol. 28, no. 7, pp. 956–965, Jul. 2009.
[38] W. Steiner, "An evaluation of SMT-based schedule synthesis for time-triggered multi-hop networks," in *Proc. of RTSS*, Dec. 2010, pp. 375–384.

Florian Sagstetter (M'13) received the Dipl.-Ing. degree in electrical engineering from Technical University of Munich, Germany, in 2010 where he is currently working towards the Ph.D. degree.

He is currently a Research Associate at the Technical University of Munich (TUM) Campus for Research Excellence and Technological Enterprise (CREATE) Centre for Electromobility in Singapore. He is working in the field of embedded systems in the automotive domain with a focus on time-triggered systems and schedule synthesis.

Peter Waszecki (M'13) received the Dipl.-Ing. degree in electrical engineering from Technical University of Munich, Germany, in 2010 where he is currently working towards the Ph.D. degree.

He is currently a Research Associate at TUM CREATE in Singapore and works in the area of embedded systems. His main research interests lie in automotive E/E architectures as well as reliable systems for electric vehicles. In particular, he is working on diagnosis methods for distributed safety-critical systems.

Sebastian Steinhorst (M'11) received the Ph.D. degree in computer science from Goethe-University Frankfurt/Main, Germany in 2011.

He is a senior researcher with TUM CREATE in Singapore. His research focuses on architectures and design automation for networked embedded systems with emphasis on electric vehicles. He is also leading a research team on embedded battery management. Before joining TUM CREATE in 2011, Sebastian was with the Electronic Design Methodology Group of Goethe-University Frankfurt/Main, Germany.

Martin Lukasiewycz (M'11) received the Ph.D. degree in computer science from the University of Erlangen-Nuremberg, Germany, in 2010.

He is currently a Principal Investigator at TUM CREATE in Singapore. Since 2014 he is Adjunct Assistant Professor at Nanyang Technological University in Singapore. Before, he worked at AUDI AG in Germany in the E/E architecture and FlexRay division, the chair Hardware/Software Co-Design at the University of Erlangen-Nuremberg in Germany, and the Institute for Real-Time Computer Systems at the Technical University of Munich.

Samarjit Chakraborty (SM'15) received the Ph.D. degree from ETH Zürich, Switzerland, in 2003.

He is a professor of electrical engineering at Technical University of Munich, where he holds the chair for Real-Time Computer Systems. Prior to joining Technical University of Munich, he was an assistant professor of computer science at the National University of Singapore from 2003 to 2008. He works on various aspects of system level design and analysis of embedded systems and has more than 150 publications in this area.