Automating the design flow for distributed embedded automotive applications: keeping your time promises, and optimizing costs, too

by
Matthias Büker   Werner Damm   Günter Ehmen
Alexander Metzner   Ingo Stierand   Eike Thaden
Automating the design flow for distributed embedded automotive applications: keeping your time promises, and optimizing costs, too

Matthias Bürker¹, Werner Damm², Günter Ehmen², Alexander Metzner³, Ingo Stierand², and Eike Thaden¹

¹ OFFIS, Germany matthias.bueker,eike.thaden@offis.de
² University Oldenburg, Germany damm,ehmen,stierand@informatik.uni-oldenburg.de
³ University of Applied Sciences Regensburg, Germany alexander.metzner@hs-regensburg.de

Abstract. We address the complete design flow from specification models of new automotive functions captured in Matlab-Simulink to their distributed execution on hierarchical bus-based electronic architectures hosting the release of already deployed automotive functions. We propose an automated design space exploration process resulting in a cost-optimized extension of the existing target hardware and an allocation of balanced task structures automatically derived from the specification model on this modified target hardware which is sufficient to guarantee both system-level timing requirements and deadlines extracted from the Matlab-Simulink specification model.

1 Introduction

We are aiming at automating significant parts of the design flow in the following typical scenario for embedded application development in automotive: given the electronic architecture $A$ of a particular car model, we are looking for a conservative cost-optimized extension of this architecture to implement a new customer feature $F$. We consider typical hierarchical bus-based target architectures, where a backbone TDMA based bus is connected via bridges to local cluster buses, such as CAN, Flexray, MOST. The modifications we strive for are conservative in that they preserve the allocation of already existing automotive functions on ECUs, and either strive for allocating the new function on existing ECUs, or, as second priority, investigate whether upgrading a controller will be sufficient to host the new functions, or, as last resort, introduce new ECUs in the local clusters. The bus-structure as such is not touched, but upgrades of local cluster buses are taken into account when upgrades of processing resources fail to yield a solution. Our design space exploration process takes into account both the costs for such modifications and timing requirements, and finds a cost-optimized extension of the existing target architecture and an allocation of the tasks of the new functions such that all timing constraints are met. We assume that $F$ is given as a Matlab-Simulink model and propose an automated process to derive a balanced task structure serving as input to the design space exploration process.

The above process bridges the time gap from system-level timing requirements and abstract timing annotations in specification models often relying on the so-called synchrony hypothesis, and the actual response times of tasks chains in such distributed target architectures, which are impacted by all of the following: worst-case execution times for individual tasks (which are increasingly hard to determine using static analysis techniques due to among others multi-issue dynamic pipelining and complex memory hierarchies increasingly involving shared cache accesses), and delays due to sharing of computation and communication resources taking into account industry standard scheduling techniques and bus protocols, this both for local cluster and inter-cluster communication. Increasingly stronger methods for deriving safe upper bounds for worst-case execution times have been developed [3, 46, 47]. Pioneering work from Burns and Tindell showed how to lift early research on schedulability analysis for single processor applications [30] under priority-driven preemptive scheduling to distributed target architectures connected by a token
Models for scheduling analysis of CAN bus and FlexRay have been provided in [38, 13]. Commercial tool offerings such as SymTA/S [24, 40] support compositional timing analysis as well as open tool boxes based on the real-time calculus [26]. In [36] we have demonstrated automatic synthesis of deployments of task structures on such target architectures, where both the allocation decisions and priorities are computed using an extension RTSAT of a SAT solver with a response-time calculus engine, and also using linear programming [18]. In [15] we have shown how to bridge the gap between system-level timing requirements expressed as timed live-sequence charts [11] for task-chains running on such distributed target architectures by deriving safe approximations of their response times as a particular simple class of timed automata. Work on design space exploration has mostly focused on automatic synthesis of deployments, assuming — in contrast to our work — a fixed target architecture. Specifically, the framework of SymTA/S [22] considers system architectures similar to ours, consisting of subsystems connected by a backbone bus. Subsystems are explored by a genetic algorithm that follows multiple optimization criteria to find Pareto-optimal solutions. Solutions of subsystems are combined with the help of a sensitivity analysis. In [29] a user-controlled genetic algorithm is proposed, that finds Pareto-optimal solutions for task allocation on fixed multi-processor architectures. Here, the analysis is based on a mixture of real-time calculus and simulation. A very flexible approach with respect to supported scheduling schemes has been proposed in [39]. It provides task and message allocation considering event-triggered and time-triggered task activations with hierarchical scheduling. Simulation based approaches for validating timing requirements in distributed targets include the RTBuilder from Geensoft and [19] where design space exploration is embedded into the Sesame framework using simulation techniques for system analysis. Design space exploration for flexible architectures is typically cast into the setting of genetic algorithms, such as in [33]. In contrast to their approach, we address incremental design processes with pre-allocated applications, and support established standards for scheduling communication resources.

In principle, the design space exploration problem can be formulated as an integer linear programming problem, where variables encode allocation decisions of tasks and messages, priorities, and processor variants are all determined in solving the optimization problem of finding a cost-minimal extension of the architecture meeting timing requirements. However, based on previous [18] and ongoing investigations in using column-generation [2], we consider the complexity of this problem beyond the scope of such exact approaches, and thus rely on heuristics to decompose the design space, alas at the price of not being able to guarantee cost-minimal solutions. Our design space exploration approach employs a combination of heuristic approaches for reduction of the design space, and analytic approaches for finding exact optimal solutions within the restricted design space. Thus cost-optimal solutions are only found within the restricted design space generated from the heuristics.

Specifically, we exploit the hierarchical structure of the target architecture and split our algorithm into three phases:

1. Global approximative analysis
2. Exact cluster-level analysis
3. Backtracking

In global approximate analysis, we use a formal analysis model for the global bus (in this paper restricted to be TDMA based bus such as from TTech or the static segment of FlexRay), current load of cluster ECUs and backbone bus, and the task network to be allocated, and use a variant of the Kernighan/Lin algorithm to partition the task networks into subnetworks and allocate these on local clusters based on estimates of their realizability, while assuring feasibility of scheduling the resulting inter-cluster messages. We automatically synthesize local deadlines based on relative computational weights of tasks, thus partitioning global system timing requirements to cluster specific deadlines.

The exact cluster-level analysis computes a cost-minimal extension and a deployment of tasks on cluster ECUs, and deployments of intra-cluster and extra-cluster messages on the local bus and either returns with a cost-minimal conservative extension of the cluster architecture meeting timing requirements, or provides a minimal subset of these which is not allocatable (called the “odd
set”), together with exact response times of allocated tasks. If the system is locally infeasible, then either the local bus is saturated, or no upgrade of existing cluster ECUs nor the possible addition of an ECU would suffice to ensure local schedulability. We allow designers to constraint the costs of upgrades to further reduce the design space. This step is performed using an extension of the RTSAT engine [35, 36] supporting architecture modifications.

A task network estimated to be feasible for execution during global analysis can be refuted during the exact local analysis step due to the following approximations. To not loose potential solutions, approximate global analysis uses worst-case execution times for tasks based on the best available of microprocessors employed in the cluster. Share-based approximations for ECU load are employed for estimating local feasibility. And, secondly, communication within local clusters is neglected.

The backtracking step uses the local response time results of allocated tasks and the odd-set, and recomputes local deadlines taking into account exact local response times for allocated tasks, and tries to reallocate the odd-set to the originally proposed cluster based on refined local deadlines. If this fails, we analyze whether upgrades of the local cluster bus allow to place the odd-set on the proposed cluster. If also this fails, we re-partition the odd-set and iterate the process. We show that this algorithm terminates in at most $O(|S| \cdot (|F| + |M|))$ steps where $|S|$ is the number of clusters in $A$, $|F|$ is the number of tasks in $F$ and $|M|$ is a typically small number of allowed upgrade for local buses, with a solution guaranteeing timeliness requirements which is cost-minimal in the considered design space, or the statement that an allocation of the new function $F$ is not feasible even when using the most expensive conservative extension of the existing architecture $A$.

Our approach offers the following advantages. First, in contrast to research cited above, it supports incremental development preserving the structure of the previous version of the electronic system architecture, thus drastically reducing the implementation effort. It fits naturally into Autosar based development processes, where the existing system- and ECU configuration can be taken as input to the design space exploration. Secondly, each step of our algorithm yields intermediate solutions, which can be again expressed in the Autosar approach and thus can be assessed by system designers, allowing to integrate manual interventions based on designer experience. Third, due to its high degree of automation, it allows early assessments of costs of adding a new feature to a car; cost-bounds can be given as targets allowing to further cut the search space.

This research on design space exploration is complemented by a formal approach of deriving tasks structures optimized for distributed execution from Matlab-Simulink models. The challenge of this step lies in the inherent mismatch between the synchronous execution model underlying Matlab-Simulink and the heterogenous target architecture typically combining both synchronous and asynchronous communication and computation models. To tackle this challenge, we define an observation semantics of Matlab-Simulink models based on interface objects. Implicitly, such notion of observability induces a partial order semantics of Matlab-Simulink models, in which all re-ordering of signal changes are considered legal, as long as causality between observable interface objects is preserved. We then enforce such synchronization constraints between tasks automatically generated from Matlab-Simulink models, and optimize the task structure while maintaining causality constraints by balancing computational load and communication load.

This research can be seen as an elaboration of research on desynchronization of synchronous models as pushed in the community of synchronous languages, e.g. by Benveniste et al [5], which compute similarly a minimal set of synchronizations which must be preserved during distributed execution. Our approach is directly integrated into Matlab-Simulink, automatically maintains relevant observation points, and exploits the partial order in optimizing the balance of the task structure.

As an interface between this front-end process and the design space exploration process, we use function networks, which provide a formal semantic based framework expressive enough to represent different communication and execution paradigms and all timing related aspects [8]. In contrast to frameworks for heterogeneous modeling such as Metropolis [12] and Ptolemy [7], our model is intended as intermediate language for real-time analysis only, but is more succinct than these in capturing the timing characteristics.
Outline The proposed design process depicted in Figure 1 starts from a specification model given in terms of a Matlab Simulink model. A function network is obtained which forms the common language for the rest of the process. In order to (partly) obtain the timing behavior of the system, worst-case execution time calculation takes place. We use a combination of well known techniques like static program analysis [20] and measuring, but this is not subject of this article. The process shown in the upper part is discussed in Sections 3 and 4, up to task creation where functions are merged to get more balanced systems. We formalize the design space to be explored in the subsequent part of the process, and the respective optimization goals in Section 5. Here, sets of architectures are defined by a set of allowed modifications of components. The design space exploration process is discussed in Section 6, followed by some considerations about termination. Section 7 concludes the paper.

Fig. 1: Process Flow

2 Extended Task Networks

Like other task network formalisms, function networks [8] are directed graphs where nodes represent processing elements, or tasks, and edges represent channels transmitting events between nodes. A channel may transport different events, or more precisely, a channel $c$ transmits events from a set $\Sigma_c$. Channel $c_3$ in Figure 2 for example transmits events from the set $\{d_1,...,d_n\}$.

Additionally, event sources allow to model events sent by the environment to the network. In Figure 2, they are depicted as rectangles with filled circles. Event source $\phi_3$ represents for example a distance sensor delivering values $d_1,...,d_n$. The occurrence of events is defined in terms of event streams [41]. Standard streams define for example periodic or sporadic event occurrences. Event streams for function networks are defined by a fixed tuple $(\Sigma^{out}, P^-, P^+, J, O)$ of parameters (not shown in the figure). $P^-$ and $P^+$ define an interval $[P^-, P^+]$ of periods for the occurrence of events at the source, $J$ is the jitter, and $O$ is an initial offset for the event source. This definition allows to model various standard event models. Events are typically randomly selected from $\Sigma^{out}$ for each firing of the event source.

The connection between nodes and channels is realized by ports representing the observation points in the system. Ports describe which events flow into and out of the corresponding nodes. Activation of nodes is captured by their input ports (small white circles). An input port activates a node whenever at least one event has occurred at all of its incoming channels. Node $f_v$ in Figure 2
for example is activated when both an event \(v_1\) on channel \(c_1\) and \(v_2\) on \(c_2\) occurs. A node having multiple input ports is activated on the activation of any of its input ports. Node \(f_m\) for example is activated for any event on channel \(c_4\), and for any event of channel \(c_5\). The combination of multiple ports and multiple input channels allows modeling of complex node activations. This is however not new, and can be found e.g. in [41].

Function nodes are sensitive to incoming events. To this end, nodes employ internal state-transition systems. Depending on the current state, an incoming event is sent to any number of output ports (small black circles). Node \(f_c\) for example sends incoming events \(v\) representing the speed of the car to port \(p_8\) leading to \(f_e\), as well as a braking event \(v\) to \(p_9\) in case the distance sensor identifies a critical distance (crit). It is also allowed that incoming events do not lead to an output at all. Node \(f_c\) for example does not send an event if the input distance was \(ok\). Each activation causes a delay for processing, depending on the input event, the current state, and the particular output port. Delays are taken from intervals with best-case and worst-case bounds. For example, an event crit that activates node \(f_c\) in state \(s_0\) needs between 4 and 5 time units to be sent to port \(p_8\), while the break event \(b\) needs between 3 and 4 time units to be send to port \(p_9\).

**Definition 1 (Basic Function Network).** A (basic) function network is a tuple \(BFN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F})\) where:

- \(\Sigma\) is a finite alphabet. Events are tuples \(e = (\sigma_1, \ldots, \sigma_k)\) of finite length with \(\sigma_i \in \Sigma\). \(\hat{\Sigma} \subset \Sigma^*\) is a finite set of events.
- \(\mathcal{P} = \mathcal{P}^I \cup \mathcal{P}^O\) is a set of input and output ports,
- \(\mathcal{C} \subset (\mathcal{P}^O \times \mathcal{P}^I)\) is a set of channels, where each output port is connected to exactly one channel,
- \( \Phi \) is a set of source nodes \( \phi = (EP, P^{out}) \) where \( EP = (\Sigma^{out}, P^-, P^+, J, O) \) is an event pattern, \( \Sigma^{out} \subseteq \Sigma \) are the events transmitted by the source, \([P^-, P^+]\) is a period interval, \( J \) is a jitter, and \( O \) is an initial offset, all being naturals, \( P^- > 0 \).
- \( P^{out} \subseteq P^O \) is a set of output ports.
- \( \mathcal{F} \) is a set of function nodes \( f = (P^{in}, A, P^{out}) \) where \( P^{in} \subseteq P^I \) is a set of input ports such that for each \( p \in P^{in} \) exists at least one incoming channel, and \( P^{out} \subseteq P^O \) is a set of output ports.
- \( A = (S, s_0, T) \) is a timed transition system where \( S \) is a non-empty finite set of states, \( s_0 \in S \) is the initial state, and \( T \) is a transition function

\[
T : P^{in} \times \hat{\Sigma} \times S \mapsto ((P^{out} \times \hat{\Sigma}) \cup \{\perp\} \rightarrow \mathbb{N}^+ \times \mathbb{N}^+) \times S
\]

mapping combinations of ports, incoming events and states to output specifications and successor states. An output specification maps tuples of output ports and events to delay intervals. Symbol \( \perp \) denotes “no output port”.

Specification languages such as Matlab Simulink often allow for explicit modeling of data objects and data access to capture not only control flow but also data flow. In order to preserve semantics when translating such specifications into function networks, being able to model data flow is an important feature. Function networks capture internal task states and complex execution patterns based on event values, which can be employed to model also data access. We can define for example function nodes modeling storage of variables, and also FIFO buffers.

To simplify modeling of data flow, we however want to be able to model explicit data storage. Furthermore, we would like to explicitly distinguish between function nodes and signals for communication between tasks. To this end, the function network formalism is extended by data nodes, that are special function nodes modeling certain data objects. Different types of data nodes as persistent ones like shared variables and FIFO buffers are defined, and volatile ones like signals.

A further enrichment is the introduction of a new channel type named \textit{read} channel. While common (activation) channels model control flow and cause an activation at their target function node, read channels model data dependencies, that is, reading access by a function node to a data node at the activation of that node. Read channels are depicted as dotted arcs and can be represented using the basic function network definition as shown in Figure 3. At the left side, a function node \( f \) reads data from two data nodes \( d_1 \) and \( d_2 \) at activation. The coordination of the read process is hidden in a complex port \( p \) which is in fact a function node as shown at the right side of Figure 3.

Fig. 3: Read Channel
stored event when it is requested via an output port. The (simplified) internal transition system shown at the bottom of Figure 4 thus contains one state per event that may be stored. Here, the state is changed if a new event \( a \) or \( b \) is written. Depending on the state \( s_a \) or \( s_b \), the respective event is returned when a read request \( r \) occurs.

A *FIFO* buffer (see Figure 5, upper left) stores a bounded queue of events and returns them in FIFO order if requested. In contrast to the shared data node, here an event may only be read once. Hence, the transition system contains one state for each possible order and length of the event queue. At the bottom of Figure 5 an example transition system for a FIFO with 2 places is depicted. If any input event \( a \) or \( b \) occurs, it is stored and the state is changed to represent the current queue state. A read event \( r \) returns the “oldest” event and changes the state accordingly. In principle, different ways exist to deal with buffer under- and overflow for example by introducing an error state in case of an overflow. In the context of this paper, an empty buffer sends a special \( \epsilon \) event when reading from it, and ignores incoming events in case of full buffers.
In Figure 6 a signal data node is shown that doesn’t store any events permanently but immediately forwards them to all output ports. This results in a transition system with a single state and one transition for each incoming event.

Another important data node type is a finite source, which produces an initial event at its output port at system startup while emitting the next event not before it has received an event at any input port. This node type is used to model cycles in function networks and is depicted in Figure 7. Its semantics is very similar to those of pre-allocated events in task networks with cyclic dependencies [25]. At the first activation by an event e from the event source, a predefined initial event is sent by the function node. After that, an event b_i is emitted at each output port for each event received on the input ports, depending on the input event a_i. Further events from the event source are ignored.

The last extension concerns communication via channels. Most task network formalisms model communication by tasks that are allocated to buses instead of processors. This approach allows to employ common analysis techniques for both computational and communication resources. For the intended use cases of function networks it is however often desired to abstract from communication. To represent communication delays in a more abstract way, function networks thus allow to annotate delays for channels. Later on, when function networks become deployed and analyzed, these communication delays are replaced by those without delays, and additional signals capturing the desired delays.

These extensions lead to the following definition of an extended function network:

**Definition 2 (Extended Function Network).** A (extended) function network is a tuple $\mathcal{FN} = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, D)$ where $(\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F} \cup D)$ is a basic function network, except:

- $C = \mathcal{CA} \cup \mathcal{CR} \subseteq (\mathcal{P} \times \mathbb{N}^+ \times \mathbb{N}^+ \times \mathcal{P})$ is a set of channels $c = (p^{\text{out}}, \delta, p^{\text{in}})$ where $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ is a delay interval, $c \in \mathcal{CA}$ are activation channels and $c \in \mathcal{CR}$ are read channels leading exclusively from data nodes to function nodes.

- $D$ is a set of data nodes $d = (\mathcal{P}^{\text{in}}, \delta, b, \mathcal{P}^{\text{out}})$ where $\mathcal{P}^{\text{in}} \subseteq \mathcal{P}^{\text{in}}$ is a set of input ports such that for each port $p$ exists exactly one incoming channel in $\mathcal{CA}$, $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ is a delay interval, $b \in \{\text{FIFO, Shared, Signal, FSource}\}$ is a data node type, $\mathcal{P}^{\text{out}} \subseteq \mathcal{P}^{\text{out}}$ is a set of output ports.

- Each input port of any $f \in \mathcal{F}$ has at least one channel from $\mathcal{CA}$, and no output port has a channel from $\mathcal{CR}$.

It may be worth to note that any extended function network can be uniquely transformed into a basic function network.
3 From Specification Models to Extended Task Networks

According to our design process, function networks are obtained from a Matlab Simulink model by translation as shown at the top part of Figure 1. The structure of the function network is obtained from the structure of the specification model by a set of transformation rules. Blocks are translated to function nodes and signals are translated to channels. The main challenge is here to translate a synchronous specification into a message-based task model. This is realized by deriving a semantics for Matlab Simulink models based on updates of signal values and relating this to events in the function network. For signals between blocks of different update frequencies the Matlab Simulink concept of Rate Transition Blocks is mapped to function networks. Timing information are obtained by deriving event patterns for specific nodes in the function network. Finally, the internal behavior of the specification is translated into executable code by employing (existing) code generators such as TargetLink [1]. Worst case execution times (WCET) are calculated for the resulting code blocks, which are used to assign weights to the respective function nodes.

For this task, we consider executable Matlab Simulink specification models that can be simulated and for which code can be generated. Hence, for translation we refer to models that meet the TargetLink modeling guidelines [17, 34]. This means among other things that only a subset of block types is available, block priorities are not considered, and algebraic loops must not be used.

3.1 (Timed) Synchronous Block Diagrams

To represent Simulink models, we pick up the basic idea of [31, 32, 37] where these models are defined as Timed Synchronous Block Diagrams.

A Synchronous Block Diagrams (SBD) consists of a set of blocks having input and output ports. Blocks are either atomic or macro (i.e. composite) blocks. Blocks are classified as either combinational (state-less) or sequential (containing internal states). Sequential blocks are called Moore-sequential if their output only depends on the state but not on the input. Diagrams are formed by connecting output ports with input ports of blocks via signals. Output ports can be connected to more than one input port while input ports can only be connected to one output port. If a block is connected to a boolean signal called trigger it is only executed if its trigger is true. For each output of a triggered block the user has to specify initial values that are valid for the starting phase before the trigger was true for the first time. An SBD is called flat if it only contains atomic blocks.

**Definition 3 (Semantics of SBDs).** A signal \( x \) is a total function \( x : \mathbb{N} \rightarrow V_x \) where \( V_x \) is a set of values. \( x(k) \) denotes the value of \( x \) at time instant \( k \). Let \( A_x \) be the block that produces \( x \), \( A_x(k) \) the result of the execution of \( A_x \) at time instant \( k \), and \( v \) the initial value of \( x \). If \( A_x \) is triggered by a signal \( t \), \( x(k) \) is determined as follows:

\[
x(k) = \begin{cases} 
  x(k) = v & \text{if } t(k) = false \land k = 0 \\
  x(k - 1) & \text{if } t(k) = false \land k > 0 \\
  x(k) = A_x(k) & \text{if } t(k) = true
\end{cases}
\]

If \( A_x \) has no trigger, \( x \) is determined as if the trigger was always true.

A Timed Synchronous Block Diagram (TBD) is an SBD where each non-triggered block has a special trigger called firing time specification (FTS). An FTS is represented as a pair \((p, init)\) where \( p \) is the period and \( init \) is the initial phase with \( init < p \). The semantics are equivalent to SBDs, because FTS are a special case of triggers. In Simulink, an FTS is called sample time.

3.2 Translating Simulink

For translation, the TBD is flattened as described by Lublinerman [32]. The hierarchy of macro blocks (e.g. subsystems in Simulink) is nevertheless used to generate constrains for the task creation
process as described in Section 4. As in [32], we assume the TBDs to be acyclic in that all cycles must contain at least one Moore-sequential block such as a “Unit Delay” block in Simulink. This induces a partial order on the firing of blocks which guarantees that the values of all signals are updated correctly.

The translation relies on some data abstraction that maps concrete values to events. For simple cases with finite value domains, this might be trivial. If the value domain is otherwise infinite, or for the behalf of simplification, other methods can be used as known from e.g. model-checking [10, 9], timing validation [46] and data flow analysis [43]. In our context, the concrete value domain of a TBD is mapped to an abstract event domain for the function network representation such that the abstract domain maintains all control-flow paths of the concrete domain. Thus, in the resulting function network, the transition systems of function nodes represent the abstract behavior of Simulink blocks obtained by their generated code. How such an abstract domain can be efficiently obtained from c-code generated from models like synchronous block diagrams has been shown in [6].

We define some auxiliary functions needed for translation. First, we describe how a firing time specification is translated into an event pattern:

Definition 4 (FTS Translation). Let $FTS = (p, init)$ be a firing time specification and $x$ be a signal. The corresponding event pattern is defined as $\text{EP}(FTS, x) = (x, [p, p], 0, init)$.

Furthermore, we need something we call FTS Extension where an FTS is extended to a given period. This means, that we represent the same FTS as a set of FTS’s with a different period comparable to expanding a fraction in mathematics.

Definition 5 (FTS Extension). Let $FTS = (p, init)$ be a firing time specification and $p_{ex} = n \times p$ ($n \in \mathbb{N}^+$) be the period it should be extended to. FTS extension is defined as:

$$\text{ex}(FTS, p_{ex}) = \{(p_{ex}, (init + i \cdot p) \mod p_{ex}) \mid 0 \leq i \leq n\}$$

Finally, we define the difference between two FTS’s and the element-relation between FTS’s.

Definition 6 (FTS Difference). Let $FTS_a = (p_a, init_a)$ and $FTS_b = (p_b, init_b)$ be firing time specifications.

$$FTS_a \setminus FTS_b \iff \text{ex}(FTS_a, p_b) \setminus \{FTS_b\}$$

Definition 7 (FTS Element). Let $FTS_a = (p_a, init_a)$ and $FTS_b = (p_b, init_b)$ be firing time specifications with $p_a = n \times p_b$ ($n \in \mathbb{N}^+$).

$$FTS_a \in FTS_b \iff FTS_a \in \text{ex}(FTS_b, p_a)$$

A flat TBD is translated to a function network as follows.

Translating Blocks Each combinational or sequential block $b$ with $n$ inputs and $m$ outputs is translated to a function node $f_b$ with one input port and $m$ output ports. All input channels are synchronized at the input port to activate the function node only when all needed inputs have been computed i.e. the appendant events have been sent. The transition function is determined as follows: For each possible combination of input events (and states) a transition is created that emits the relevant output events on the respective output ports. The events are obtained by an abstraction of the concrete signal values as discussed before. The delay of each transition is
defined by determining the worst case execution time of this block assuming the input configuration described by the events.

Each Moore-sequential block \( b \) with \( n \) input signals and \( m \) output signals is translated to a \( FSource \) data node with one input port and \( m \) output ports.

Each Rate Transition Block with an input signal from a block \( a \) and an output signal to a block \( b \) is translated to a special function node \( f_{rt} \) that converts the sample time \( FTS_a = (p_a, init_a) \) of block \( a \) to the sample time \( FTS_b = (p_b, init_b) \) of block \( b \). We assume that \( p_a \) and \( p_b \) are integer multiples i.e. either \( p_a = n * p_b \) or \( p_b = n * p_a \) (\( n \in \mathbb{N}^+ \)). This is assured by selecting the Simulink check box “Ensure deterministic data transfer” of the rate transition block. In fact, this also would ensure that \( init_a = init_b \), but to be more general we only assume \( init_a \leq init_b \) here. The function node \( f_{rt} \) has an incoming channel at its input port \( p^i_a \) from \( f_a \) (function node corresponding to \( a \)) and an outgoing channel from its output port \( p^o_b \) to \( f_b \) (function node corresponding to \( b \)).

The characteristic of \( f_{rt} \) depends on the following:

1. If \( p_a > p_b \), we add an input port \( p^i_a \to f_{rt} \) triggered by a \( Source \) node \( \phi_{rt} \) with an event pattern that implements \( FTS_a \setminus FTS_b \). Furthermore, we add a \( Shared \) data node that is written from a second output port \( p^o_w \) and has a read channel to \( p^i_a \). This data node is necessary to store the latest event received at \( p^i_a \). The transition system then contains a transition from \( p^i_a \) to \( p^o_w \) and from \( p^o_w \) to \( p^o_b \). If furthermore \( FTS_a \in FTS_b \), there exists also a transition from \( p^i_a \) to \( p^o_b \). If otherwise \( FTS_a \notin FTS_b \), there exists no such transition, because block \( a \) and \( b \) never run at the same time-step in Simulink.

2. If \( p_a \leq p_b \) and \( FTS_b \notin FTS_a \), this induces that the phase difference \( \Delta = init_b - init_a \) is always a multiple of \( p_b \) (otherwise \( FTS_b \in FTS_a \) would not hold). In this situation, the output port of \( f_{rt} \) to \( f_b \) must only fire if both blocks \( a \) and \( b \) are triggered. This is realized by creating a transition system that acts as a kind of period multiplier i.e. it only sends events each \( \frac{p_a}{p_b} \)-th while the other transitions do not fire any output port. This transition system has the states \( s_0, ..., s_{k-1} \) where \( k = \frac{p_a}{p_b} \) (\( k \in \mathbb{N}^+ \)). There exists a transition from \( s_{k-1} \) to \( s_0 \) that fires an event at the output port \( p^o_b \) and there exist transitions from \( s_i \) to \( s_{i+1} \) with \( 0 \leq i \leq k-1 \) that do not fire any event. The initial state is \( s_{k-1} \) with \( j = \frac{\Delta}{p_a} \).

3. If \( p_a < p_b \) and \( FTS_b \notin FTS_a \), we do the same as in case 1) for \( FTS_b \notin FTS_a \).

For each Data Store Memory block a data node \( d \) of type Shared is created. For each corresponding Data Store Write block with an incoming signal from a block \( w \), there exists an input port at \( d \) with an incoming channel from \( f_w \). Accordingly, for each Data Store Read block with an outgoing signal to block \( r \), there exists an output port at \( d \) with an outgoing channel to \( f_r \).

For each Sink block a Shared data node is created and the corresponding input signal is translated to an activation channel.

Translating Triggers An FTS that triggers a block \( b \) with no inputs from other blocks uses a boolean signal \( x \) is translated to a source \( \phi = (EP(FTS,e),\{p^{out}\}) \) and an activation channel from \( p^{out} \) to the input port \( p^{in} \) of \( f_b \). A trigger \( t \) leading from a port \( o \) of block \( a \) to a block \( b \) is translated to an activation channel from the respective port \( p_o \) of \( f_a \) to the input port of \( f_b \).

Translating Signals Let \( x \) be a signal leading from output port \( o \) of block \( a \) to an input port of block \( b \). Let further be \( p^o_o \) the output port \( o \) is translated to. Then \( x \) is translated as follows:

- If \( a \) and \( b \) have the same sample time i.e. \( FTS_a = FTS_b \), \( x \) is translated to an activation channel from \( p^o_o \) to \( p^o_p \).
- If \( a \) and \( b \) have different sample times i.e. \( FTS_a \neq FTS_b \), we create a virtual rate transition block between \( a \) and \( b \) whose translation was described before.

If \( b \) has a trigger (that is not an FTS), the FTS of \( b \) is equal to the FTS of its triggering block (and vice versa). Additionally, we define the data size \( DataSize(c) \) for each created channel \( c \) by considering the data type of the appendant Simulink signal. This is used to define weights in the subsequent task creation process.
3.3 Translating Stateflow

Due to a missing confirmed semantics of Stateflow, we rely on scientific approaches to define semantics such as [23] and [42]. In the latter one, a safe subset for Stateflow is defined that precludes unbounded behavior by avoiding loops in any graph of junctions and transitions, permitting the use of the backtracking mechanism of flow transitions and not allowing any inter-level transitions. We restrict to Stateflow models that satisfy this safe subset and the TargetLink modeling guidelines [34]. In particular, we assume that the models are designed to not depend on the graphical model layout (e.g. the 12 o’clock rule) to solve non-determinism.

Targeting at transition systems of function nodes, Stateflow blocks are translated by building the flattened parallel composition of all its Statecharts. In the current state of work, transitions can only be translated if the trigger event is an input signal, the condition refers only to input signals, and actions only refer to output signals.

While transitions between states can be translated straightforward, transitions that involve junctions are translated by summarizing them to abstract transitions that again lead from state to state. This is done by creating for each path in a subgraph that only consists of transitions and junctions an abstract transition. The events and conditions of this abstract transition are each the conjunction of the events and conditions of all original transitions of that path. The same is done with the actions. This step can only be done with the assumption that backtracking does not take place or that no transition has any condition action but only transition actions. Additionally, there must be no two actions on one path that consider the same output signal. An extension of this approach would be to also allow internal events, conditions and actions as well as additive actions on the same output signal. This is however subject to future work.

Those abstract transitions can be translated in the same way as “ordinary” transitions between states as described in the following.

Translating Transitions Let $TS$ be a set of stateflow transitions $ts = (s,e,c \rightarrow a,d) \in TS$ where

- $s$ (source) and $d$ (destination) are states,
- $e$ is an event of a trigger $tr$,
- $c$ is a condition over inputs $i_1, ..., i_n$, and
- $a = \{(o_1 \leftarrow v_1), ..., (o_n \leftarrow v_n)\}$ is a set of actions where $(o_i \leftarrow v_i)$ denotes that the value $v_i$ is assigned to the output signal $o_i$.

To respect the testing order defined in Stateflow, we start with translating those transitions that have events and conditions. $map_T(TS)$ creates for each such stateflow transition $ts$ a function node transition $t \in T$ with $t = (p_0^n, e', c', s \rightarrow a' \cup a_0, d)$ where

- $e' = \{e\}$,
- $c'$ is the set of all tuples of (condition) events that fulfill the condition $c$,
- $a'$ is a set of output specifications $(p_{o_i}, e_i) \rightarrow \delta_i$ where
  - $p_{o_i}$ denotes the function network port where the output signal $o_i$ is translated to
  - $e_i$ is the event resulting from the abstraction of the signal configuration $o_i = v_i$
  - $\delta_i$ is defined as the worst case execution time of this block with the input configuration described by the state, event and condition of the Stateflow transition.
- $a_0$ is a set of output specifications where for each output port that is not affected by any action in $a$ a default event is sent.

Secondly, transitions are translated that only have events but no condition. Then $c'$ is the set of all tuples of (condition) events that are not part of any previously translated transition with the same Stateflow event. Afterwards, transitions with conditions but without events are translated. Here, $e'$ is the set of all Stateflow events that are not part of any previously translated transition with the same condition. For transitions without event or condition, a transition is created that contains a set of all tuples of events that are not covered by any already translated transition. If a stateflow chart is not complete, for each state a self-transition is added that is triggered if none of the existing transitions can be executed.
Translating Stateflow Blocks. For each Stateflow block $b$ with input signals $I = i_1, \ldots, i_n$, output signals $O = o_1, \ldots, o_m$, a trigger $tr$, a state set $ST$ and a set of transitions $TS$, a function node $f_b = (P_{\text{in}}^b, A_b, P_{\text{out}}^b)$ is created where

- $P_{\text{in}}^b = \{p_{i_n}^b\}$ contains one input port,
- for each output signal $o_j$ of $b$ a port $p_{o_j}^b \in P_{\text{out}}^b$ is created.
- $A_b = (S, s_0, T)$ where $S$ is similar to the flattened state set of the Stateflow block, $s_0$ is the initial state of $b$ after flattening, and $T = \text{map}_T(TS)$.

State Actions are translated by adding them to the affected function node transitions i.e. “entry actions” of a state $s$ are added to all transitions leading from $s' \neq s$ to $s$, “exit actions” of a state $s$ are added to all transitions leading from $s$ to $s' \neq s$, and “during actions” of a state $s$ are added to all self-transitions leading from $s$ to $s$.

3.4 Preserving Semantics

For both translation and the following task creation, preserving semantics of the original specification is a key issue. Matlab Simulink models are inherently untimed, where block executions and communication are instantaneously, and are not affected by delays and jitter due to variances in the delays. Obviously, nothing of this holds for any implementation of a Matlab Simulink model that runs on real hardware. Moreover, TBD models follow the synchronous paradigm while function networks are asynchronous models based on communication by message transfer. If we translate one paradigm to the other, we have to take care that (i) execution ordering of blocks and (ii) for each block execution the currently available inputs match those of the original semantics.

To bridge this what we call time gap, we start by associating block executions with observable events in the corresponding function network. An execution of a TBD can be defined by a (infinite) sequence of updates on signals that occur due to block executions. The time instances at which a block is executed depends on the parameters of the block trigger. At each execution, the block reads the current values from its input signals, and updates its output signals. Function networks combine node executions and reading input values by the concept of events. A function node is executed whenever all channels on an input port of the node have sent an event. Note that events also serve as (abstract) values. We relate the two semantics in that we associate an update of a signal in the TBD to the occurrence of an event in a function network. Referring to Definition 3, an update for example of signal $x$ by a block $A_x$ at time instant $k$ occurs if $A_x$ has been triggered, and the new value of $x$ is defined by $A_x(k)$. In the corresponding function network, the execution of $f_{A_x}$ causes an event $A_x(k)$ to be sent to the output port $p_x$.

We can now reason about semantics preservation by showing that all possible sequences of events occurring in the function network coincide with sequences of updates of signals in the TBD. Two aspects are of interest here, namely covering of (untimed) sequences of signal updates by corresponding event sequences, and timing. Concerning time, we have to relax requirements to time instances of particular event occurrences due to delays caused by execution and communication of an underlying hardware architecture. We will come to this back later.

Concerning sequence-covering we can observe, that the translation guarantees that for each block execution a corresponding event is sent in the function network. Without going into detail we propose that the translation also preserves all aspects of the value domain relevant with respect to control flow. Furthermore, the translation preserves the structure and connectivity of the TBD. However, TBDs generally allow to execute a block without executing a subsequent block. For the corresponding function network, this would result in an event sent by the first block, that possibly results in an activation of the second one. According to [14] and [32], the structure of a TBD imposes a partial order of block executions. Following [14], connectivity among blocks with the same sample times induces a partition of blocks called synchronous sets. Hence, the semantics of a synchronous set is defined by all execution sequences of blocks of the same synchronous set satisfying the induced partial order in terms of signal updates.
Fig. 8: Preserving Synchronous Sets of Matlab Models

The translation maintains the partial order of blocks and thus preserves the order of signal updates by the corresponding events in the function network. For blocks within the same synchronous set, all input channels of the corresponding function node are synchronized equivalently to the input signals of the set. Each time a block is executed, its output signals are updated which is represented in the function network by producing an event on each output port of a function node for each activation.

For Stateflow blocks this is valid as well because here again each transition emits an output event at each output port, even if there is no appendant action in the Stateflow transition. Activation of successor blocks in the TBD is preserved by translating the corresponding signals to appropriate activation channels.

The translation enforces that different synchronous sets are either coupled by (implicit) rate transition blocks, or are not connected at all. For rate transition blocks between different synchronous sets, a specific function node \( f_{\text{rt}} \) is created that imposes the same behavior. If the rate transition leads from a block \( a \) to a block \( b \) where \( a \) has a greater period than \( b \), then block \( b \) requires its input data more often than it is produced by block \( a \). Thus, the rate transition block stores the latest values of the output signal of block \( a \) and offers it to block \( b \) whenever it is needed. In the function network, this is realized by adding another \( \text{Source} \) node that triggers \( f_{\text{rt}} \) whenever it is not triggered by the preceding function node \( f_a \) but an activation is required by the successor \( f_b \) due to its sample time. The received events are stored in a \( \text{Shared} \) data node and forwarded at the next activation by the source. The \( \text{Source} \) produces events with an event pattern that implements \( FTS_a \setminus FTS_b \), which results exactly in the set of sample times that is needed to compensate the mismatch between \( FTS_a \) and \( FTS_b \). Furthermore, if a different initial phase delay leads to the situation that \( a \) and \( b \) never run in the same simulation step (which is not allowed for a fixed-step solver), \( f_{\text{rt}} \) only produces output events if it is activated by the newly created \( \text{Source} \) but not if activated by \( f_a \). Otherwise, output events are produced in both cases. \( FTS_a \setminus FTS_b \) already considers these situations.

In Figure 8 an example Simulink model is depicted at the top and the function network resulting from translation at the bottom. The block \( RTB \) represents a rate transition that leads from a sample time \( ST_1 = [6,0] \) (depicted in blue) to a sample time \( ST_2 = [2,0] \) (depicted in green). In the function network representation this is realized by a function node named \( RTB \) with a source node that implements \( ST_2 \setminus ST_1 = \{[6,2],[6,4]\} \) and a shared data node storing input events.

If block \( a \) has period \( p_a \) which is smaller than period \( p_b \) of block \( b \), block \( b \) must only be activated every \( k \)th time where \( k = \frac{p_a}{p_b} \). Note that \( k \) is also an integer because periods are always integer multiples. This is realized in the function network by creating the appendant transition system as described in section 3.2. If the initial phases differ in a way such that \( a \) and \( b \) never are executed in the same simulation step, we proceed as in the case discussed before.

As the final step to capture Matlab Simulink semantics correctly when the corresponding implementation is executed on a platform, we have to ensure that all functions of a synchronous
set are executed within their associated sample time. This is both a sufficient and necessary condition to preserve partial ordering induced by the original Matlab Simulink model, because it is the weakest condition to avoid overlapping activations of blocks and function nodes, respectively. Note that the frequency of block executions is maintained by the event sources in the corresponding function network.

To capture such constraints, we employ the concept of deadlines which is natural in the context of real-time systems. A deadline $D \in T$ defines a maximum delay between certain events $e_1, e_2$ like activations or completions of function nodes. If we take an arbitrary execution of a function network, let $t_i(e_1)$ be the time instant of the $i$th occurrence of its input event $e_1$, and $t_i(e_2)$ for its output event $e_2$, respectively. We say the deadline is satisfied if and only if $|t_i(e_2) - t_i(e_1)| \leq D$ for all $i \in \mathbb{N}$, and all such executions.

Timing constraints are given in terms of end-to-end deadlines, which define upper bounds for executions of task chains. A task chain is the set of nodes on all paths from some start node to some end node (cf. Figure 9). In the following we restrict to end-to-end deadlines for simple linear task chains. More complex chains (for example where the control flow is split and joined again) are divided into sets of linear chains, which is always possible. The corresponding deadline reasons about the time span between the activation of the first node and finishing the execution of the last node of the chain, which coincides with sending the respective output event. For example, all tasks corresponding to the synchronous set $ST_3$ in Figure 8 due to translation have to be executed within 5 time units, which results in two end-to-end deadlines.

For an implementation of a Matlab Simulink model to maintain semantics, we thus have to define end-to-end deadlines for the nodes of each synchronous set. The length of the deadline is defined by the period of the set. Note that for the following task creation process it is important to maintain the different synchronous sets because otherwise no valid deadlines could be defined. Suitable constraints defined for the task creation algorithm forbid to merge any two consecutive function nodes of different synchronous sets into the same task.

Timing constraints might not only be defined due to constraints from the underlying model, but may also be other system level requirements. For the programming of control loops it is for example crucial to restrict the execution length of the control algorithm for stability reasons. The process is indeed able to deal with such system level requirements too. For more complex requirements than simple deadlines, such as specifications of protocols, it would easily be possible to integrate more involved analysis techniques into the process commonly employing the formalism of function networks [8].

4 Task Creation - Semantics and Application

A function network that results from translation of a Matlab Simulink model is typically unbalanced, consisting of a large number of nodes with high variance in computational weights (weights describe proportional amounts of capacity nodes consume on a computation resource). If we treat each node as a single task, this would result in a large communication overhead when tasks are spread over the distributed hardware resources in the following design exploration process. Together with the fact that many of these tasks would have relatively small execution times compared to task-switching times, this seems not to be a very promising way to obtain an optimal system at all. Accordingly, we want to obtain more suitable task sets by merging nodes. This is what we call Task Creation.

In order to obtain suitable tasks sets, it has to be decided which nodes shall be merged and which not. The default Matlab Simulink strategy when generating code with Realtime Workshop is to put all blocks with the same sample time into the same task. In a function network, this would mean to partition all nodes with the same period and offset into one task. Often, these are connected blocks that cannot be executed concurrently because they depend on each other.
But also blocks of completely independent block chains are merged to one task by this strategy. This may lead to very “big” tasks when there are many blocks with identical sample times. For simulation purposes, this is quite useful because all these blocks are executed at the same simulation step, execution times are neglected, and parallel execution is not regarded at all. For the execution on a distributed real-time systems, this strategy precludes any of these blocks from being executed concurrently, which might lead to deadline violations.

Nevertheless, two connected nodes with the same activation pattern are still good candidates to be executed in the same task, because they cannot be executed concurrently anyway. Additionally, communication between tasks on different ECUs is very expensive in a distributed system, because it takes much more time to communicate via a bus than via local (shared) memory. Furthermore, often the bus is the bottleneck of such systems and can hardly be upgraded. So, one objective for task creation for such systems should be to minimize communication between tasks.

But we also need to consider the computational weight of a node which means the load it produces on an ECU. It is not useful to create tasks with arbitrary large weights because they may be either not executable on some ECUs, or they would reduce the number of possible schedules due to large blocking times. On the other hand, tasks should not be too lightweight, because the sum of task switching times would increase and waste a significant amount of ECU capacity.

From the perspective of the design space exploration process, it is desirable to have tasks with balanced weights. This would largely reduce the impact of computational density of tasks, and the decision where to allocate a task would be more driven by the actual optimization criteria such as communication or costs. In other words, assuming a perfectly balanced task network without any communication between tasks would induce that the allocation decision can be made with a maximum of freedom and flexibility in the design space exploration.

In summary, communication density between tasks should be minimized to relieve the bus, and the weights of tasks should be balanced to keep the allocation process flexible. To achieve this, we introduce a metric called cohesion.

4.1 Cohesion and Weights

Task creation partitions the nodes of a function network, and merges the nodes within the resulting partitions to get a set of tasks. We however do not want to partition only function nodes but also data nodes, because in the following exploration process also data nodes such as shared variables and buffers have to be allocated on ECUs. For deployment, we can think of a data node as a particular piece of code implementing the respective data object. Communication between nodes is represented by the channels connecting them. Formally, task creation partitions the sets $F$ and $D$ of a function network into a task set $T = \{\tau_1, \ldots, \tau_n\}$ where $\tau_i = \{n_{i,1}, \ldots, n_{i,k}\}$, $n_{i,j} \in F \cup D$. The communication structure of the resulting task set is then determined by the channels of the corresponding nodes.

As outlined before, the task set shall be chosen such that communication density is minimized and node weights are balanced. Node balancing is achieved by minimizing the variance with respect to the average task weight. The variance can only be decreased by merging two nodes whose sum of weights is closer to the average weight than their single weights. Nodes with large weights can only become more distant from the average.

This leads to the definition of cohesion: Nodes are attracted by high communication density, and distracted by high node weights. This is achieved by preferably merging nodes with low weights, while communication is minimized by considering the sum of all weights of channels between nodes of different partitions. For the definition of cohesion, we introduce some weight factors $\alpha, \beta > 0$ that are adjusted by user preference to control the process:

$$\text{cohesion}(FN, T) = \alpha \cdot \bar{w}(T) + \beta \cdot \text{com}(C')$$
where

\[
\tilde{w}(\mathcal{T}) = \frac{1}{n} \cdot \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\overline{w}(\mathcal{T}) - w(\tau_i))^2}
\]

\[
\overline{w}(\mathcal{T}) = \frac{1}{n} \cdot \sum_{i=1}^{n} w(\tau_i) \quad \text{(average task weight)}
\]

\[
w(\tau_i) = \sum_{n_{i,j} \in \tau_i} w(n_{i,j}) \quad \text{(weight of task } \tau_i)\]

\[
com(\mathcal{C}(\mathcal{T})) = \sum_{c \in \mathcal{C}(\mathcal{T})} com(c) \quad \text{(comm. weights)}
\]

The set of channels \( \mathcal{C}(\mathcal{T}) \) to be considered for the communication weight depends on the respective task set and contains all channels that connect nodes in different task partitions.

The weight \( w(n) \) of a node depends on its execution times and activation patterns while execution times strongly depend on the compiler target. Thus, we assume that in the following exploration process all potential target processors are known and the WCET for each transition of a node on each potential processor can be estimated. This issue is however out of scope of this work. We define the weight of a transition as the minimum WCET among all potential processors of the target architecture. We choose the minimum because without further knowledge about possible allocations we assume that each node will be allocated to its best fitting processor. If however any constraint for a node exists that forbid the allocation to a certain set of processors, this set is excluded from weight calculation.

More precisely, the weight of a node is defined as the sum of its port weights. The weight of a port is the maximum weight of all transitions with an input on this port, divided by the ports lower period bound. The period depends on the event pattern of preceding nodes and event sources. Weight for function node \( f = (P^{in}, A, P^{out}) \) is defined as:

\[
w(f) = \sum_{p_i \in P^{in}} w(p_i)
\]

where

- \( w(p_i) = \frac{1}{P_{i}^-} \cdot \max(w(t_{i,j})) \) is the weight of input port \( p_i \)
- \( w(t_{i,j}) \) the weight of the \( j \)th transition with input port \( p_i \)
- \( P_{i}^- \) is the lower period bound of \( p_i \).

For weight calculation of data nodes, we assume their “implementation” as discussed for basic function networks.

Communication density is defined in terms of weights for channels. The weight of a channel \( c \) depends on its data size, the communication rate, and the maximum bandwidth in bytes/s of all local buses \( k \) that are known a priori. It is defined as:

\[
com(c) = \frac{\text{DataSize}(c)}{\text{max Bandwidth}_k} \cdot \frac{1}{P_{c}^-}
\]

where

- \( \text{DataSize}(c) \) is the data size of \( c \),
- \( P_{c}^- \) is the lower period bound of the channel.

The period of a channel can in general be retrieved for example by so-called event stream propagation for task networks. For Matlab Simulink models, the period of any communication as well as any port activation is always well-defined.

Beside the optimization goal of minimizing the cohesion function, there exists a user controlled set of constraints restricting the task creation process. First, we assume a minimum \( (n^-) \) and maximum \( (n^+) \) number of tasks. Second, we introduce minimum and maximum achievable task weights. The minimum task weight is intended to counteract thrashing caused by tasks with too small execution time, which induce frequent task switching and thus low processor utilization.
The maximum task weight ensures, that there is sufficient potential parallelism, thus allowing to reduce end-to-end latencies. As for $\alpha$ and $\beta$, these parameters will typically highly depend on the respective application and are given by the user.

Further constraints can be obtained from the hierarchical structure of the original specification model which is a Matlab Simulink model here. For example, the user may claim that all blocks of certain subsystems have to be in the same task and are thus executed on the same ECU. The choice of such subsystems may be done by restricting to subsystems of specific hierarchy-levels (e.g. only the two deepest levels) or by manual selection. Another kind of constraint may be to forbid that blocks of certain subsystems are partitioned into one task.

4.2 Syntax and Semantics of task creation

In this section, we elaborate on the question what task creation actually means and which semantic consequences it implicates. The process of task creation is divided into three independent operations: merging of function nodes, elimination of local data nodes and elimination of self-activations.

The first operation is the step that is mandatory for task creation because it describes the actual merging of two function nodes into one. The second and third operation are optional and can be considered as optimisation functions. The elimination of local data nodes reduces complexity and lowers communication time by removing data nodes that are used exclusively by one function node and whose clearance does not affect the execution semantics. When eliminating self-activations this does not only reduce complexity and communication time but also reduces the set of possible execution traces by concatenating two consecutive activations to one activation.

In the following, the different operations are defined with the help of a component concept. Here, a component is a part of a function network with a well-defined interface to the remaining function network. The component contains all nodes and channels that are relevant for the operation. The interface is defined by the ports connected to nodes outside of that component. Each operation replaces one component by another component with an identical port interface. For semantical correctness of an operation, the causality i.e. the partial order of interface events has to be maintained. This also holds for internal component events as long as they remain observable when applying the operation. If an event is no longer observable, causality is preserved by maintaining the control flow.

**Merging nodes** When two function nodes are merged this involves a restructuring of the function network by replacing a component of two function nodes $f_1$ and $f_2$ by a component with one function node $f_{1+2}$ which models the same behaviour. This means that the sets of input ports of $f_1$ and $f_2$ are unified. The same is done for the output ports. The transition system of $f_{1+2}$ is obtained by building the usual parallel composition $\parallel$ of the transition systems of $f_1$ and $f_2$.

**Definition 8 (Node Merging).** Let

$\quad FN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ be a function network and

$\quad f_1 = (P_{in}^1, A_1, P_{out}^1) \in \mathcal{F}$ and $f_2 = (P_{in}^2, A_2, P_{out}^2) \in \mathcal{F}$ be two function nodes.

The merge operation is defined as follows:

$\quad \odot (FN, f_1, f_2) = (\Sigma, \mathcal{P}, \mathcal{F} \setminus \{f_1, f_2\} \cup \{f_{1+2}\}, \Phi, \mathcal{D}, \mathcal{C})$

where $f_{1+2} = (P_{in}^{1+2} \cup P_{in}^2, A_1 \parallel A_2, P_{out}^{1+2} \cup P_{out}^2)$

The semantic consequences of merging two function nodes $f_1$ and $f_2$ is mainly that $f_1$ and $f_2$ from now on are executed on the same scheduling resource i.e. transitions of $f_1$ and $f_2$ cannot be executed concurrently even this was possible before merging. Thus, the set of possible execution traces might be reduced. But even though we restrict function network behavior by merging the two nodes, causality is still preserved in terms of partial ordering of events. This is because all events, ports, channels and data nodes are maintained as well as the transition systems of the
original function nodes. Therefore, also the partial order of all events (including the interface events) is preserved.

Concerning timing, node merging may enlarge the delay between the arrival of an event at an input port and the emitted output event, because transitions that could be executed concurrently before cannot be executed concurrently after merging. Because computational weights of function nodes are calculated by the sum of their port weights and all ports are maintained together with their transitions, the weight of $f_{1+2}$ is the sum of the single weights of $f_1$ and $f_2$ as claimed in the cohesion function.

![Fig. 10: Merging Nodes](image)

On the left side of Figure 10 a component of a function network with two function nodes $f_1$ and $f_2$ is depicted where $f_1$ triggers $f_2$ via a signal node and two activation channels (activation path is depicted in green). Furthermore, there are read and activation channels to a shared data node. The channels with loose endings at the component interfaces indicate connections to further function network elements outside of the considered component.

The same function network part after merging nodes $f_1$ and $f_2$ is depicted on the right side of Figure 10. The green activation path is now a self-activation i.e. $f_{1+2}$ activates itself at a different input port. The shared data node is unaffected and the read channel moves with its target port to the node $f_{1+2}$.

It is quite obvious that the merging operation is associative, because both the joining of ports and the parallel composition of transition systems is associative. This becomes important for the application of this operation in the task creation algorithm.

Please note, that this operation can also be applied on two function nodes that are directly connected by an activation channel (with a delay $\geq 0$) instead of a Signal data node. The result would look quite similar to Figure 10 while the green parts would be replaced by a single activation channel.

**Elimination of Local Data Nodes**

A second operation is the elimination of local data nodes. A data node $d$ is local to a function node $f$, if this is the only function node connected to $d$ via channels and it is in the same task partition as $f$. Under the condition that the behavior of $f$ does not depend on the read events from $d$ this data node can be eliminated without violating causality of the remaining events.

When performing this operation, the data node and the corresponding read and write channels are removed. The transition system of $f$ is modified such that the respective events are removed from any transition. Additionally, the output port that writes to $d$ is removed and each transition that emitted events at this port now writes the $\bot$ symbol instead.

**Definition 9 (Data Node Elimination).** Let

- $FN = (\Sigma, P, C, \Phi, F, D)$ be a function network,
- $f = (P_{\text{in}}, (S, s_0, T), P_{\text{out}}) \in F$ a function node and
- $d \in D$ a data node with an incoming activation channel $c_w = (p_w, \delta_w, p_d) \ (p_w \in P_{\text{out}})$ transmitting event $w$ and an outgoing read channel $c_r = (p_d, \delta_r, p_r) \ (p_r \in P_{\text{in}})$ transmitting event $r$. 


The data node elimination function is defined as follows:

\[ \ast_d(FN, f, d) = (\Sigma', P', F, \Phi, D', C') \]

where

- \( \Sigma' = \Sigma \setminus \{r, w\} \), \( P' = P \setminus \{p_d, p_{d'}, p_w\} \),
- \( D' = D \setminus \{d\} \), \( C' = C \setminus \{c_r, c_w\} \) and
- \( f = (P^\text{in}, (S, s_0, T'), P^\text{out} \setminus \{p_w\}) \) where \( T' \) contains all transitions from \( T \) while
  - each occurrence of \( p_w \) in a transition is replaced by \( \bot \),
  - the event \( r \) is deleted in each transition whose event expression \( E \) contains \( r \).

The consequences of eliminating a data node \( d \) are the following: The data node is removed with all its ports and the connected channels. Additionally, the respective output port of the function node that writes to this data node is removed. The involved write and read events are removed as well. Under the assumption that the behavior of the function node does not depend on the read event of the removed data node, this operation has no consequences on the causality of the remaining events. All input ports of the function node are obtained together with all activation events of that node. The transitions of the function node are maintained as well while they are cleaned by the read event \( r \). Thus, the partial order between input and output events of the components interface is still valid.

Concerning timing, the delay between any input and output signal that involved the reading of event \( r \) becomes smaller because the data is now available locally at the function node and the time for reading the event (possibly over a bus) is saved. Thus, any end-to-end deadlines that were valid before this operation are still valid after it.

![Eliminating local data nodes](image)

**Fig. 11: Eliminating local data nodes**

In Figure 11 on the left side a component is shown with a function node and a local data node that is eliminated on the right side. The green arrows in the function node indicate the affected transitions to show that these are maintained even if an output port is removed.

**Elimination of Self-Activations** The third operation is the elimination of self-activations of a function node \( f \). Self-activations are self-loops from an output port to an input port of \( f \) either via a Signal data node or an activation channel with a delay \( > 0 \). While self-activations could of course be part of any function network, they particularly arise when two function nodes with an activation dependency are merged. That’s why their elimination is a typical continuation of the node melting process. The consequences of the elimination of self-activations are, that the (possibly) involved data node is removed if it is not used by other function nodes. Additionally, channels may be deleted including the appendant ports and events.

To be able to apply this operation without violating causality of events, the input port that is the target of the self-activation loop must not have any other incoming read or activation channels. A further necessary condition for a loop containing a data node \( d \) is, that it must not have both incoming and outgoing channels to other function nodes than \( f \). In this case, it would not be possible to remove the self-activation without affecting activations from or to other nodes.

Before defining the operation for eliminating self-activations itself, we need some help functions to be defined before. The first one adds an output delay to a given set of output specifications of a transition. An output specification is a pair of a port and an event mapped to a delay interval.
The self-activation elimination function is defined as follows:

**Definition 12 (Self-Activation Elimination).** Let \( \psi = \{(p'_1, E'_1 \rightarrow \delta_1), \ldots, (p'_n, E'_n \rightarrow \delta_n)\} \) with \( \delta_i = [\delta_i^-, \delta_i^+] \) be a set of output specifications of a transition and \( \delta = [\delta^-, \delta^+] \) a delay interval.

\[ \boxplus(\psi, \delta) = \{(p'_1, E'_1 \rightarrow (\delta_1 + \delta)), \ldots, (p'_n, E'_n \rightarrow (\delta_n + \delta))\} \] where \( \delta_i + \delta = [\delta_i^-, \delta_i^+ + \delta^+] \)

Next, we define how a given transition system changes when a self-activation via an output port \( p_w \) and an input port \( p_a \) is eliminated. For each transition that does not contain one of these two ports nothing changes. But all pairs of transitions that would execute successive in the case of a self-activation need to be concatenated.

This means, that the left part (input port, input event, origin state) of the first transition becomes also the left part for the concatenated transition. The right part of this new transition must not fire events at \( p_w \), if this port is removed during self-loop elimination. Instead, the delays of the affected events are added to the delays of all output ports of the second transition. All other output specifications remain unchanged.

**Definition 11 (Self-Transition Concatenation).** Let \( T \) be a transition system, \( p_a \) be an input port and \( p_w \) an output port of a self-activation.

\[ \boxplus(T, p_a, p_w) = T' \] where

1. \( \forall t = (p, E, s \rightarrow \Psi, s') \in T \) where \( p \neq p_a \) and \( \hat{\psi} \psi = (p_w, E_w \rightarrow \delta_w) \in \Psi \) holds: \( t \in T' \)

2. For each pair of transitions
   - \( t_1 = (p_1, E_1, s_1 \rightarrow \Psi_1, s'_1) \in T \) where \( \exists \psi_w = (p_w, E_w \rightarrow \delta_w) \in \Psi_1 \) and
   - \( t_2 = (p_a, E_2, s_2 \rightarrow \Psi_2, s'_2) \in T \) holds:
     \[ \exists t' \in T' \text{ with } t = (p_1, E_1, s_1 \rightarrow \Psi_1 \setminus \psi_w \cup \boxplus(\Psi_2, \delta_w, s'_2)) \]

The elimination of self-activations is defined for a function node \( f \) that activates itself via a signal data node \( d \). This is the more general case compared to activations by single activation channels that are covered as well as a simplification of the first case considered in the definition. The self-activation is resolved by replacing it by a set of concatenated transitions. This means that subsequent executions that occur with the self-activation are merged into one using the previously defined functions.

**Definition 12 (Self-Activation Elimination).** Let

\( - \text{FN} = (\Sigma, P, C, \Phi, F, D) \) be a function network,
\( - f = (P^{in}, (S, s_0, T), P^{out}) \in F \) a function node and
\( - d = (P^{in}, \delta_d, b_d, P^{out}) \in D \) a Signal data node that has an incoming activation channel \( c_w = (p_w, \delta_w, p_d) \) (\( p_w \in P^{in} \)) transmitting event \( w \) and an outgoing activation channel \( c_a = (p_a, \delta_a, p_a) \) (\( p_a \in P^{in} \)) to \( f \) transmitting event \( a \).

The self-activation elimination function is defined as follows:

\[ *_a(FN, f, d) = (\Sigma', P', C, \Phi, D', C') \]

while we get different results depending on how \( d \) is connected to other function nodes. We distinguish the following cases:

1. If \( d \) has no other channels than \( c_w \) and \( c_a \), then
   \[ - \Sigma' = \Sigma \setminus \{w, a\}, P' = P \setminus \{p_d, p_a, p_w\}, D' = D \setminus d, C' = C \setminus \{c_w, c_a\}, \]
   \[ - f = (P^{in}, (S, s_0, \boxplus(T, p_a, p_w)), P^{out} \setminus \{p_w\}) \]

2. If \( d \) has an additional activation channel to another function node, then
   \[ - \Sigma' = \Sigma \setminus \{a\}, P' = P \setminus \{p_a, p_w\}, D' = D \setminus c_a, \]
   \[ - f = (P^{in}, (S, s_0, \boxplus(T, p_a, p_w)) \cup T_w), P^{out} \setminus \{p_w\} \text{ where } T_w = \{(p, E, s \rightarrow \Psi, s') \in T \mid \exists (p_w, E_w \rightarrow \delta_w) \} \]
   \[ - d = (P^{in} \setminus \delta, P^{out} \setminus p_d) \]

3. If \( d \) has an additional activation channel from another function node \( f_2 \), then
\[ \Sigma' = \Sigma \setminus \{w\}, \mathcal{P}' = \mathcal{P} \setminus \{p_d, p_w\}, \mathcal{D}' = \mathcal{D}, \mathcal{C}' = \mathcal{C} \setminus \{c_a\}, \]
\[ f = (\mathcal{P}_{in}, (S, s_0, \sqcap(T, p_a, p_w) \cup T_a), \mathcal{P}_{out} \setminus \{p_w\}) \text{ where } T_a = \{(p, E, s \rightarrow \Psi, s') \in T \mid p = p_a\} \]
\[ d = (\mathcal{P}_{d in} \setminus p_d, \delta, \mathcal{P}_{d out}) \]

The semantical consequence of this operation is mainly the change of causal event chains that include the events \( w \) and \( a \). All these event chains are shortened by removing the sub-chain from \( w \) to \( a \). This is realized by concatenating the appendant transitions. But even if these events are removed completely, the causality of the remaining observable events of the component is still preserved.

![Fig. 12: Eliminating self loop (local signal)](image)

This is exemplified in Figure 12 where a function node with a self-activation is shown whose involved data node is local to that function node. The red and green arrows in the function node indicate two transitions that are executed successively (not necessarily direct successively) where the red one is the first and the green one is the second transition. On the right side the situation is shown after the ports 2 and 3 were removed by eliminating the self-activation. Here, the red and green transitions are concatenated. But an activation at port 1 still leads to an event at port 4 as on the left side. What is different is the fact that both transitions are now executed as one transition. While on the left side it was possible that another activation occurs between these transitions it is not possible on the right side anymore. This means that the set of execution traces is reduced with this operation.

![Fig. 13: Eliminating self loop (signal only has incoming channels from \( f_1 \))](image)

Figure 13 shows another example where the involved data node has a further outgoing activation channel to another function node \( f_2 \). Thus, the data node is still existent after self-loop elimination but the back-loop channel to \( f \) is removed. Additionally, the output port 2 still exists to activate \( f_2 \). So, even if the red and green transitions are concatenated to one transition, the firing of port 2 is maintained. This keeps the causality of the interface events to \( f_2 \).

Concerning timing, the delay between any input interface event and output interface event either stays the same (if it is not affected by the self-activation) or is even shortened because the delay of the self-activation (which is always \( > 0 \)) is no longer existent. Furthermore, the number of task switches is reduced by this operation because two activations are now executed as one.

### 4.3 Task Creation Algorithm

The objective of the task creation algorithm is to partition the function nodes into a set of at most \( n^+ \) partitions while minimizing the cohesion function. The function nodes of the same partitions
are later merged to one task by the previously defined operation. From the semantic point of view each two function nodes may be merged without violating any causality of events. Deadlines can be checked approximately for a certain partitioning by considering the sum of delays on the longest path from the start to the end point. Communication delays are considered only if they are global i.e. they lead from one partition to another.

We define a two-step approach where first an initial solution is created by a constructive algorithm that is specialized to the present problem. Afterwards, a state-of-the-art algorithm is applied to optimize the initial solution.

To be able to prepare an appropriate initial partitioning, the properties of the systems we regard here have to be considered. The function networks that are derived from specification tools like Matlab Simulink typically consist of several synchronous sets that are chains of function nodes that share the same activating event pattern. These chains may be synchronised to serve as inputs for central controller nodes which then again trigger outgoing node chains. But still some chains may be completely independent allowing concurrent execution. This leads to a network with a high amount of communication and very few isolated nodes.

With respect to this structure, the algorithm for the initial partitioning is communication-driven as well. Thus, it does not consider each possible combination of function nodes as merging candidates but only those that induce communication between different task partitions either via direct channels or via data nodes and channels. Considering the cohesion function, this mainly reduces the communication in the system while respecting the balance of task weights. At the same time, the parallelism in the system is obtained because nodes without any (indirect) communication are not merged. The algorithm works as follows:

1. Put each function node into a separate partition,
2. For each data node $d$ connected to function nodes:
   - Obtain all partitions $T_1, ..., T_k$ connected to $d$,
   - Check if merging of $\{T_1, ..., T_k\}$ is valid w.r.t to constraints,
   - Calculate gain of cohesion by joining $\{T_1, ..., T_k\}$,
3. Do the same as in 2) for each channel $c$ connecting two function nodes,
4. Choose the solution with the best gain,
5. Proceed until either
   (a) no further valid set of merging candidates with positive gain can be found,
   (b) no data node $d$ and no channel $c$ is available anymore connecting different partitions.

The result of this first step is a system where the communication between tasks is reduced to the minimum allowed due to constraints and the node-balancing paradigm in the cohesion function. But the network is still not necessarily well balanced because concurrent function nodes where not considered as merging candidates yet. Additionally, the given minimum task weight may not be given for each partition.

To resolve this, we apply in a second step a combination of the well-known Kernighan/Lin [27] and Fiduccia/Mattheyses [21] algorithms to address primarily the balancing of node weights. This algorithm tries to move or exchange nodes between partitions while now also nodes without any communication may be grouped together if this lowers cohesion. Some discussion about complexity and optimality of these algorithms can be found in section 6.1.

The result is a set of partitions of function and data nodes while all function nodes of the same partition are merged into one function node to create a task. The order in which the nodes are merged is irrelevant, because the node merge operation $\odot$ is associative. Empty partitions can be ignored and do not result in a task. As a further step and to complete the task creation process, it is checked for each local data node and each self-activation if it can be eliminated with the appendant operation. Here, local data nodes have to be eliminated before self-activation elimination because local data nodes may induce read channels that prevent a semantic-preserving self-loop elimination. Even though both elimination operations are not absolutely necessary for task creation, they play an important role to reduce task switches and communication times. Furthermore, eliminated data nodes and the appendant channels have no longer to be considered in the following design exploration process.
5 Design Space and Objectives

Given a Task Network \( TN^* = (\Sigma^*, P^*, C^*, \Phi^*, T^*, S^*) \) obtained by task creation, the objective is to find a satisficing system architecture that provides a feasible allocation for \( TN^* \). The definition of a task network follows the one of a basic function network, except for \( T^* \) and \( S^* \). \( T^* \) is a set of function nodes now called tasks. Furthermore, we assume the task network to have signals, which greatly simplifies the following optimization process. If the original function network does not contain signals, this is obtained by replacing each channel by two channels with a signal in between. The new signal inherits the weight of the original channel, and the weights of the new channels is set to 0.

In our setting, the design space is a set of possible system architectures. We restrict to systems common in automotive industry (cf. Figure 14):

- A backbone bus connects a number of local clusters. Currently we restrict to bus systems using some TDMA protocol such as the static part of FlexRay.
- Each cluster contains a number of processors, connected by a local bus. One processor is the gateway to the backbone bus.
- Typical processor types are ARM, TriCore, and PowerPC.
- Typical local bus types are CAN, TTP, and FlexRay.

Naturally, the design space also subsumes all possible allocations for tasks and signals in \( TN^* \) to system architectures.

We generally assume an existing system that shall be extended, that is, a system architecture \( A \) and a pre-allocated task network \( TN \). We then allow addition and modification of hardware components (processors, buses), which defines the design space. The “existing” system may be empty.

5.1 Design Space

Formally, a system \( M = (TN, A, \xi) \) is defined by:

1) A pre-allocated task network \( TN = (\Sigma, P, C, \Phi, T, S) \),
2) A system architecture \( A = (P, B, C) \) where \( P = \{p_1, ..., p_n\} \) is a set of processors, \( B = \{b_1, ..., b_m\} \) is a set of buses, and \( C \subseteq P \times B \) is a set of connections. \( A \) is assumed to satisfy topological restrictions explained above:
   - Processors belong to clusters with one local bus.
   - One processor of each cluster is also connected to the global bus acting as a gateway.
We refer in the following often to clusters \( S_1, ..., S_n \) of \( A \) which are in fact subsets of the components of \( A \).

Processors \( p_i \) and buses \( b_j \) are tuples of the form

\[
p_i = (Type, ClockRate, MemoryLayout, ..., Costs),
\]
\[
b_j = (Type, Bandwidth, ..., Costs)
\]

where the elements are mainly used for WCET analysis.
3) An allocation is defined as \( \xi(TN,A) \subseteq (T \times P) \cup (S \times B) \). Allocations are assumed to be consistent:
- Task allocations are unique for each task.
- Signals may be allocated to multiple buses where each allocation is modeled by one bus message.
- Topologies of task network and architecture match.

A design space is then defined as follows:
- The task network \( TN^* \) that is added to \( TN \),
- A set of system architectures \( A = \{A_1, ..., A_s\} \), derived from \( A \), and
- A set of allocations \( \Xi = \{\xi(TN^*,A_i) \cup \xi(TN,A_i) \mid A_i \in A\} \).

The allocation of the existing task network is assumed to be fixed, except changes of the target processor or bus type.

The basic idea to obtain \( A \) is to define all allowed modifications of the existing system. A modification is a tuple \( m = (p,p') \), or \( m = (b,b') \) that maps a specific hardware component to another, so we write \( m(p) = p' \). We then define a set of modification rules:

1. Allowed hardware modifications. e.g.
   - Processor \( p_2 \) may be upgraded to 150MHz:
     \[
     m(p_2) = m((ARM7,100MHz,...,5E)) = (ARM7,150MHz,...,7E)
     \]
   - A processor of type PPC may be added:
     \[
     m(p_9) = m((-,-,...,0E)) = (PPC,200MHz,...,15E)
     \]

2. Conflict clauses for hardware modifications, e.g.
   - Modifications \( m_i \) and \( m_j \) must not apply together.

A modification \( m = (p,p') \) maps a system architecture onto another, and modifications can be applied successively:

\[
    m(A) = A[p' \leftarrow p] = A', \quad m_i(m_j(A)) = m_i(A') = A''.
\]

Additionally, allocation rules can be specified to induce a set \( \Xi \) of task allocations:

1. Allowed task allocations:
   - Task \( \tau_i \) may be allocated on processors \( p_1, p_4 \) or \( p_7 \).
2. Conflict clauses and “must” clauses for task allocations:
   - Tasks \( \tau_i \) and \( \tau_j \) must not be allocated together.

Given a set of rules defined by the user, the design space is obtained by applying all allowed combinations of modifications, and by all allowed task allocations.

5.2 Design Objectives

To state the objectives for the exploration process more formally, we want to find for a given task network \( TN^* \) and a design space \( (A, TN \cup TN^*, \Xi) \) a system \((TN \cup TN^*, A, \xi)\) with architecture \( A \in A \) and an allocation \( \xi \in \Xi \) such that:

1. \( \xi \) is feasible for tasks and communication,
2. Costs for the architecture are minimal.
Since we are dealing with real-time systems, any allocation must satisfy timing constraints for task executions and communication in order to be feasible. Timing constraints are given in terms of end-to-end deadlines as defined in Section 3, which define upper bounds for executions of task chains. The first objective states that all such timing constraints must be satisfied.

The second objective defines the optimization goal of the process. While costs is a very general objective, in our case costs are defined in terms of costs for modifications which has been specified before as an element of the processor and bus tuples. For the sake of simplicity, we restrict in the following to monetary costs. For example, replacing an existing 20MHz processor that costs 5€ by a 30MHz processor that costs 12€ imposes costs of 7€. An already existing system does not have costs. We further assume costs for modifications to be additive. This is not a general restriction of the approach but also for the sake of simplicity. Otherwise, the cost functions described later in this context would become more complex, and less intuitive.

6 Design Space Exploration

For the design space exploration process, we propose a divide-and-conquer strategy. A global analysis (Figure 15a) partitions the task set $\mathcal{T}^*$ of $\mathcal{TN}^*$ into sets $\mathcal{T}_1^*,...,\mathcal{T}_n^*$ for allocation on the respective clusters $S_1,...,S_n$. To this end, global analysis estimates for each cluster the hardware modifications needed to get a feasible allocation for a given task partition. Estimation employs a share-based analysis, where resource capacities and task weights are calculated. Global analysis then searches for the costs optimal solution, that is a task partition with minimal induced modification costs.

Each task partition may impose communication over the backbone bus. Although task creation tries to minimize overall communication between tasks, a particular task partition found by global analysis may lead to communication over the backbone bus that exceeds available bandwidth. Global analysis thus tries to find for each task partition a TDMA slot scheme for the backbone bus that satisfies the additional communication load. Solutions exceeding available bandwidth are rejected.

Finally, global analysis also synthesizes deadlines for the task partitions that are used in the subsequent local analysis (Figure 15b). Local analysis takes the task partition proposed by global analysis and allocates the tasks (and messages) on the ECUs (and buses) of the respective clusters such that they become feasible with respect to the synthesized deadlines.

It is obviously not guaranteed that this process results in a feasible system, because the (exact) local analysis of one or more clusters could potentially show that allocation decisions made by global analysis are in fact infeasible due to the given costs constraints. The following reasons may lead to this result:

1. Global analysis estimation is share-based, and assumes that the whole remaining capacity of a processor is available for task executions. Furthermore, estimated task weights are in general only lower bounds.
2. Global analysis does not take communication on local buses into account.
3. Deadline synthesis may be insufficient, because an optimal partition of deadlines would be possible only with a-priori knowledge of task response times.

The process is thus completed by a backtracking phase (Figure 15c) that refines the decisions made in the global analysis, and then iterates the whole process. To this end, the local analysis of each cluster not only tries to find a feasible allocation, but also calculates a so-called odd-set of tasks which, if it would be removed, results in a feasible cluster. In the backtracking phase, deadlines are re-synthesized with respect to the tasks that have been successfully allocated so far. Furthermore, additional modifications for the local bus systems that may lead to feasible solutions are applied.
6.1 Global Analysis

Finding a task partition with minimal hardware modification costs can be reduced to the problem of finding a minimal cut that partitions a weighted graph into \( n \) partitions. The costs of a cut is however not calculated by edge weights but by the costs for allocating tasks on clusters. The partitioning algorithm takes task network \( TN^* \) and cost functions \( C^I_i \) for clusters \( S_i \) \((i = 1, \ldots, n)\) of the architecture, and computes a partition \( \{T_i^*, \ldots, T_n^*\} \) of the tasks in \( T^* \) for allocation on clusters with minimal costs:

\[
\min \left\{ C^I_1(x_1) + \ldots + C^I_n(x_n) \right\} \quad \text{where} \quad x_i = \sum_{\tau \in T_i^*} w(\tau).
\]

where the task weights \( w(\tau) \) depend on the cluster they are allocated to. In contrast to task creation where weights depend on the minimum WCET over all possible processors, task weights are now calculated from the minimal WCET of all processors of that cluster as depicted in Figure 16.

**Capacities and Costs** Costs calculation for allocating tasks on clusters is share-based, where we assume a task to consume computational capacities according to its weight. To this end, we need a measure of available capacities for processors. Given a time interval \([t, t+d]\), any processor under load has some fraction \( I(t, d) \) of idle time. Although \( I(t, d) \) varies over time, it becomes stable for sufficiently long time intervals. Thus, \( I := \lim_{d \to \infty} I(t, d) \) is the capacity of the processor. It can be calculated as \( I = 1 - U \) where \( U \) is the worst case utilization of the resource.

In order to relate modifications and costs, we propose that each modification provides additional resource capacity. Assume for example some processor \( p \) and a modified version \( p_m = m(p) \) with capacities \( I_p \) for \( p \), and \( I_{p_m} \) for \( p_m \), respectively. The capacity of the modification is then \( I_m = I_{p_m} - I_p \).
Calculating the costs \( C_m \) of a modification is simple; since each component of an architecture has assigned costs as defined in Section 5, the costs of a modification that adds a component is equal to the costs of the respective component. If a modification replaces a component, the costs are the difference of the two component costs.

We relate costs of modifications to the additional capacity they provide in a way that is suitable for applying the partitioning algorithm. To this end, we “sort” sets of modifications by the costs they impose. A costs function for each possible modification \( m \) in a particular cluster is defined by \( C^I_m(x) := C_m \) for \( 0 < x \leq I_m \), \( C^I_m(0) := 0 \), where \( I_m \) is the additional capacity provided by the modification. \( C^I_m(x) \) thus causes additional costs \( C_m \) for up to \( I_m \) additional load. According to the defined design space, for each processor \( p \) exists a set of possible modifications \( \{ m_1, \ldots, m_n \} \). This results in a set of functions \( C^I_{m_1}(x), \ldots, C^I_{m_n}(x) \). New components being added due to a modification get 100% capacity.

Let \( I_0 \) be the remaining capacity of \( p \) without any modifications. Since we assume that only one modification can be applied to a particular processor at the same time, we define the costs function as a simple minimum:

\[
C^I_p(x) = \begin{cases} 
0, & \text{if } x \leq I_0 \\
\min_{1 \leq i \leq n} C^I_{m_i}(x - I_0), & \text{else}
\end{cases}
\]

For a cluster \( S_i \) with processors \( p_1, \ldots, p_k \), we define the cost function \( C^I_i \) to be a folding as follows:

\[
C^I_i(x) = \min \left\{ \sum_{j=1}^{k} C^I_{p_j}(x_j) \mid x = \sum_{j=1}^{k} x_j, \ 0 \leq x_j \leq I_j \right\}
\]

Infeasible weights (that is \( x > \sum I_j \)) get costs \( \infty \). The resulting function \( C^I_i(x) \) for cluster \( S_i \) returns for an additional load \( x \) the minimal modification costs needed to provide this capacity. Note that the function does not capture which set of modifications exactly leads to the additional capacity and costs but that there is such a set for the cluster. Figure 17 illustrates an example where two cost functions (at the left side) are folded to get the costs function at the right side.

![Fig. 17: Calculating Costs Functions](image)

In order to get the costs for a particular task partition \( T^*_i \), the analysis calculates the additional capacity \( x_i = \sum_{\tau \in T^*_i} w(\tau) \) needed to allocate the tasks of \( T^*_i \). The final costs are the sum of the resulting costs \( C^I_i(x_i) \) for all \( T^*_i \).

**Deadline Synthesis and Global Bus Communication** Despite the fact that costs estimation of the global analysis may be too optimistic to get a feasible task allocation at all, we need to calculate local deadlines for each cluster in order to reason about feasibility in the subsequent local analysis.

Local deadlines for the respective clusters are synthesized by partitioning global deadlines. Given a task chain \( T = \{ \tau_1, \ldots, \tau_m \} \), it will be partitioned into sets \( T_1, \ldots, T_n \) for clusters \( S_1, \ldots, S_n \) due to task partitioning (cf. Figure 18). For the (global) deadline \( D_T \) of \( T \), and task weights \( w(T) \) and \( w(T_i) \) (which are the sums of the respective task weights), global analysis calculates in the first iteration the local deadline for those tasks of the chain allocated to cluster \( S_i \) as the respective fraction of \( D_T \):

\[
D_i = \frac{w(T_i)}{w(T)} \cdot D_T \tag{1}
\]

This calculation however neglects global bus communication. In fact, communication on the backbone bus has impact on two key elements of the exploration process:

1. **Local deadlines for the respective clusters are synthesized by partitioning global deadlines.**
2. **Deadline Synthesis and Global Bus Communication** Despite the fact that costs estimation of the global analysis may be too optimistic to get a feasible task allocation at all, we need to calculate local deadlines for each cluster in order to reason about feasibility in the subsequent local analysis.
Deadline synthesis, because time intervals needed for transmission on and synchronization with the backbone bus are no longer available for the execution of parts of a task chain on a cluster,

Schedule of the backbone, because messages have to be sent and received timely while not affecting other messages that are already using the backbone bus.

Note that we don’t want to touch an already deployed system, neither on the clusters nor on the backbone bus.

If tasks of two different partitions of $T$ need to communicate, additional load is allocated to the backbone bus. To keep simplicity we assume that for $T$, $n-1$ messages $\{\zeta_1, \ldots, \zeta_{n-1}\} = M$ have to use the backbone bus. This is however the case only if the tasks of $T$ are spread among all clusters. We denote $M$ the set of all such messages for all task chains. We further denote $M_O$ the set of already deployed messages on the backbone bus.

For the backbone bus, we restrict to a time-triggered bus that provides a set $\Delta := \{1, \ldots, N\}$ of $N$ slots of uniform length $\Lambda$, forming a cycle of length $L_{\text{TDMA}} = |\Delta| \cdot \Lambda$. The allocation problem is to find a mapping $\sigma : M \to 2^{\Delta \setminus \Delta_O}$ such that for all $\zeta_i, \zeta_j \in M$, $\zeta_i \neq \zeta_j$:

$$\sigma(\zeta_i) \neq \emptyset \land \sigma(\zeta_j) \neq \emptyset \land \sigma(\zeta_i) \cap \sigma(\zeta_j) = \emptyset$$

where $\Delta_O$ is the set of slots reserved by messages in $M_O$.

Since the backbone bus is driven by the time-triggered paradigm, a task chain may suffer synchronization delay; once a task on a cluster has finished its computation and wants to send message $\zeta$, it has to wait for one of its slots $\sigma(\zeta)$. We define an upper bound $Q$ for the time a task is allowed to wait for synchronization. $Q$ is a fraction of the available deadline for the task chain $\zeta$ belongs to, and thus part of the deadline synthesis. Since the synchronization delay is equal to the maximum distance between the slots assigned to messages, the mapping must further satisfy for all $\zeta_i \in M$ with $\sigma(\zeta_i) = \{s_1, \ldots, s_t\}$, $s_i \in \Delta$:

$$\forall s_k, s_{k+1} \in \sigma(\zeta_i) : A \cdot (s_{k+1} - s_k) \leq Q$$

where $L_{\text{TDMA}}$ is the length of the TDMA cycle.

$$\land \ L_{\text{TDMA}} + A \cdot (s_1 - s_t) \leq Q$$

Figure 19 depicts an example schedule of a backbone of length $L_{\text{TDMA}} = 20$ ($A = 1$), where the slots $\Delta_O$ occupied by $M_O$ are marked with $X$, and $M = \{A, B, C\}$ is the set of new messages stemming from three task chains. The bounds are set to $Q_A = 7$, $Q_B = 12$, and $Q_C = 20$. The following slot assignment is a feasible solution to the problem: $\sigma(A) = \{6, 12, 19\}$, $\sigma(B) = \{10, 20\}$, and $\sigma(C) = \{4\}$.

In order to capture feasibility of the global bus, each time the global partitioning procedure creates a candidate, a valid schedule of the backbone bus system is calculated, if possible. Otherwise the partition candidate is rejected.
For a feasible solution, the bus analysis provides delay times \( R_1, ..., R_{n-1} \) for messages \( M \) of task chain \( T \), where \( R_i \) is the maximum distance between any two subsequent slots for message \( \zeta_i \). Since these global communication delays are no more available for task executions, Equation (1) is modified by subtracting them from the global deadlines before calculating the remaining deadlines:

\[
D_i = \frac{w(T_i)}{w(T)} \cdot (D_T - \sum_{k=1}^{n-1} R_k)
\]

Note, that the parameter \( Q \) strongly affects the validity of the schedule of the backbone. Hence, the issue of deadline synthesis is a matter of optimization, too. Observing the impact of the value of \( Q \) for each task chain can be used to drive re-calculations during backtracking. This is however subject of further work.

While finding optimal solutions for TDMA scheduling is known to be expensive, we employ in the context of this paper a more simple heuristic algorithm that is based on a rate-monotonic scheduling approach. For message \( \zeta_i \), it starts with a deadline \( Q_\zeta \) which is a suitable fraction of the deadline of the task chain. The algorithm may return higher response times for some messages, which are bounded by some factor of the original \( Q_\zeta \). The algorithm gives up if it cannot find a solution with such deadlines. The algorithm has complexity \( O(N^3) \) and will be discussed elsewhere. It provides the solution of the example above (which is optimal) in about 1.7ms. Run-times for calculating valid schedules for the example presented in the next section are also negligible short.

Techniques and Comparison To find cost optimal task partitions, the algorithm of Kernighan/Lin [27] has been employed, where the cost function of the algorithm has been replaced as described earlier. Furthermore, the algorithm has been extended to support multiple partitions, and performance improvements borrowed from [16] have been implemented. The algorithm, which we denote \( KL^+ \), has been applied to a set of benchmarks and compared to a number of algorithms:

- Kernighan/Lin (KL) [27] starts with an initial partition, and exchanges iteratively the node pair with the highest cost reduction. Exchanged nodes are marked and the algorithm finishes when no nodes exist or no better solutions can be found. The complexity is \( O(n^2 \cdot \log(n)) \).
- Fiduccia/Mattheyses (FM) [21] is a modification of [27]. Nodes are not exchanged but moved while the partition size is bounded. The complexity is \( O(n) \).
- Quick Cut (QC) [16] is also a modification of [27]. It maintains a neighborhood relation and efficient data structures. The complexity is \( O(m \cdot \log(n)) \) with \( m \) edges.
- Simulated Annealing (SA) [28] is a probabilistic approach that considers also worse solutions to escape from local minima.

Table 1 compares the runtimes of the different algorithms. The first three columns show the number of tasks, task connections, and the number of partitions for splitting the respective task network. Entries marked with ‘-’ where aborted after 10min. It shows that the \( KL^+ \) algorithm scales well also for large systems. Table 2 compares the quality of the results for a number of task networks. The second column shows the costs of the optimal solution for the respective system \((\#T/\#E/\#P = \text{number of tasks/edges/partitions})\). The results show the costs calculated by the algorithms and how far away they are from the optimal solution. The table shows, that the variation of the results is quite high for all algorithms. This reflects the hard nature of the optimal partitioning problem. Interestingly, the (quite simple) KL algorithm performs only slightly worse than Simulated Annealing although one would expect that it is more sensitive. The \( KL^+ \) algorithm was chosen because it provides the best trade-off between optimality and performance.

6.2 Local Analysis

Local analysis is applied to find feasible task allocations for all clusters. Feasibility is defined in terms of synthesized deadlines. For the end-to-end response time of each local task chain it must hold that \( R(T_i) \leq D_i \). Local analysis has to allocate all tasks assigned to a cluster to the ECUs of
the cluster with respect to such deadlines. This is a classical real-time scheduling and deployment problem.

Additionally, the analysis algorithm optimizes the local architecture by deciding which modifications proposed by global analysis shall take place in order to achieve a feasible task allocation. In order to determine which modifications led to certain costs for cluster $S_i$, the respective costs function is assessed as depicted in Figure 20. Each level in the costs function is associated with sets of modifications $M_1, \ldots, M_n$, where each set $M_k = \{m_k, 1, \ldots, m_k, n_k\}$ imposes these costs when applied to the original system architecture. Each such set induces a modified cluster $S'_{i,k} = M_k(S_i)$, that has to be considered by local analysis.

As mentioned earlier, estimation made by global analysis may be too optimistic. This may result in infeasible clusters where not all tasks can be deployed. In this case, local analysis calculates a so-called task odd-set. Assume a partition $\mathcal{T}_i$ of tasks to be deployed on cluster $S_i$. An odd-set $O_i$ is a subset of $\mathcal{T}_i$ such that (in the following order):

1. Task set $\mathcal{T}_i \setminus O_i$ is feasible on cluster $S_i$.
2. Communication between $O_i$ and $\mathcal{T}_i \setminus O_i$ is minimized.
3. Weight $w(O_i)$ is minimized.

Two different approaches are currently under investigation for local analysis, an RTSAT based approach, and a combination of Spare-Time Analysis and MILP [44]. In the following we focus on the RTSAT approach that is more matured. RTSAT is a SAT-checker modulo theory [35, 36], where a SAT-checker is modified on specific decision points to perform classical scheduling theory. The scheduling analysis part is able to modify problem variables of the SAT-checker model to slice the search space. After an initial solution is found, binary search is used to find an optimal one. The search procedure terminates if either a solution has been found, or a predefined timeout is reached, which possibly leads to only sub-optimal solutions. A comprehensive discussion of the tool can be found in [35] and performance benchmarks in [36].
Table 2: Optimality (in %)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10/20/2</td>
<td>75  118</td>
<td>63.0 118</td>
<td>63.6 118</td>
<td>63.6 237</td>
<td>31.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15/40/2</td>
<td>272 273</td>
<td>99.0 273</td>
<td>99.6 273</td>
<td>99.6 346</td>
<td>78.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17/40/2</td>
<td>265 302</td>
<td>87.7 302</td>
<td>87.7 302</td>
<td>87.7 291</td>
<td>63.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>192 331</td>
<td>58.0 331</td>
<td>58.0 331</td>
<td>58.0 376</td>
<td>51.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>214 257</td>
<td>83.3 257</td>
<td>83.3 257</td>
<td>83.3 319</td>
<td>69.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>199 259</td>
<td>95.2 259</td>
<td>95.2 259</td>
<td>95.2 330</td>
<td>60.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20/50/2</td>
<td>271 314</td>
<td>86.3 314</td>
<td>86.3 372</td>
<td>72.8 367</td>
<td>73.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>279 280</td>
<td>99.6 280</td>
<td>99.6 280</td>
<td>99.6 374</td>
<td>74.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>245 311</td>
<td>78.8 311</td>
<td>78.8 567</td>
<td>43.2 245</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>218 327</td>
<td>66.7 327</td>
<td>66.7 327</td>
<td>66.7 356</td>
<td>61.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20/100/2</td>
<td>721 842</td>
<td>85.0 842</td>
<td>85.0 1082</td>
<td>66.6</td>
<td>911</td>
<td>79.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>832 892</td>
<td>93.3 892</td>
<td>93.3 892</td>
<td>93.3 841</td>
<td>97.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>803 829</td>
<td>96.9 829</td>
<td>96.9 829</td>
<td>96.9 836</td>
<td>96.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>846 902</td>
<td>93.8 902</td>
<td>93.8 971</td>
<td>87.1 957</td>
<td>88.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20/200/2</td>
<td>1751 1998</td>
<td>87.0</td>
<td>1998</td>
<td>87.6</td>
<td>2060</td>
<td>72.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1579 1920</td>
<td>82.2</td>
<td>1920</td>
<td>82.2</td>
<td>1920</td>
<td>82.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1711 1901</td>
<td>90.0</td>
<td>1901</td>
<td>90.0</td>
<td>1934</td>
<td>88.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1853 2077</td>
<td>89.2</td>
<td>2077</td>
<td>89.2</td>
<td>1931</td>
<td>96.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12/40/3</td>
<td>416 534</td>
<td>77.9</td>
<td>534</td>
<td>77.9</td>
<td>534</td>
<td>77.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>307 388</td>
<td>62.9</td>
<td>388</td>
<td>62.9</td>
<td>588</td>
<td>62.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>537 609</td>
<td>88.2</td>
<td>609</td>
<td>88.2</td>
<td>624</td>
<td>86.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>421 521</td>
<td>80.8</td>
<td>521</td>
<td>80.8</td>
<td>521</td>
<td>80.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>15/80/3</td>
<td>979 1050</td>
<td>93.2</td>
<td>1050</td>
<td>93.2</td>
<td>1192</td>
<td>82.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>967 1155</td>
<td>83.7</td>
<td>1155</td>
<td>83.7</td>
<td>1242</td>
<td>77.9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>878 1042</td>
<td>84.3</td>
<td>1042</td>
<td>84.3</td>
<td>1006</td>
<td>87.1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>890 929</td>
<td>93.8</td>
<td>929</td>
<td>93.8</td>
<td>929</td>
<td>93.8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12/100/4</td>
<td>1500 1600</td>
<td>93.3</td>
<td>1600</td>
<td>93.3</td>
<td>1600</td>
<td>93.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1625 1792</td>
<td>90.7</td>
<td>1792</td>
<td>90.7</td>
<td>1792</td>
<td>90.7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12/10/4</td>
<td>86 85</td>
<td>100.0</td>
<td>85 100.0</td>
<td>85</td>
<td>100.0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>12/40/4</td>
<td>473 617</td>
<td>76.7</td>
<td>617</td>
<td>76.7</td>
<td>617</td>
<td>76.7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The original algorithm has been extended for the DSE process. Firstly, SAT clauses have been added to capture system modifications and respective costs. ECUs are thus no more fixed but may change according to the sets of modifications $M_k$ derived as described before.

In order to get task odd-sets, a “virtual” ECU is added to each cluster, where tasks are allocated to only if otherwise the cluster becomes infeasible. Costs for allocating tasks to this ECU has been chosen to be very high, which leads the algorithm to avoid allocation on the ECU whenever possible. Tasks allocated to the virtual ECU on the other hand are assumed to have no impact to the task chains they belong to. Thus, local deadlines are available for all other tasks not allocated on the virtual ECU. The costs for allocating a task to the virtual ECU depend on the tasks weight and its communication to other tasks within the same cluster. Prioritizing communication and task weights is achieved by assigning according factors to communication and task weights.

Finally, communication between tasks of the cluster and other clusters is modeled by (synthetic) communication tasks allocated to the gateway ECU (the one connected to the backbone bus) of a cluster. Each communication between an application task that have to pass the backbone bus is routed via the communication tasks.

### 6.3 Backtracking

Decisions made by global analysis may lead to infeasible clusters. In particular, local analysis for one or more clusters may result in an odd-set of tasks that are removed from the task set in order to get a partial feasible allocation.

Three inaccuracies in the process can lead to this result. The first one is the fact, that synthesis of local deadlines from end-to-end deadlines is based on estimated task weights. These weights
are only lower bounds of the actual delays experienced by tasks when running on a particular ECU. Execution delays thus might be higher than estimated, resulting in insufficiently partitioned deadlines.

Secondly, global analysis neglects local task communication. Such communication imposes additional delays to task executions reducing available time. Furthermore, certain task allocations might not be possible due to bandwidth constraints on the local bus.

Finally, because global analysis deals with lower bounds for task weights, the remaining processor capacity might not be sufficient to allocate all assigned tasks.

In the backtracking phase, different measures are taken to alleviate these deficiencies. To this end, backtracking iterates three nested loops. The two inner loops modify system parameters at the cluster level involving only reprocessing of the local analysis. The outside loop then iterates the process including global analysis.

1) Re-synthesis of local deadlines. For any task chain \( T = \{T_1, ..., T_n\} \), local analysis allocates in each step a subset of tasks. Once allocated, the impact of such tasks becomes known and need not be estimated anymore.

This reduction of uncertainty is exploited by a re-calculation of local deadlines within the inner backtracking loop. To this end, the response times of all allocated tasks are subtracted from the end-to-end deadline of the task chain. Also the transmission times of the outgoing messages of the corresponding tasks are subtracted. With this new “end-to-end” deadline, deadline synthesis is applied for the remaining tasks of the task chain that are not yet allocated. The resulting local deadlines are finally extended by those response times of tasks (and transmission times the corresponding messages) belonging to the respective clusters that have been subtracted before.

Depending on the remaining tasks that could not be allocated so far, the impact of particular tasks weights to deadline synthesis might change. Thus, local deadlines for some parts \( T_k \) of a task chain might become smaller due to re-synthesis, which is for example the case for the middle cluster in Figure 21. This certainly avoids to find better task allocations for the respective clusters. Local analysis is however forced to keep tasks allocated that have been allocated before. Because deadline re-synthesis always maintains the length of local deadlines such that existing task allocations remain feasible, in each backtracking step the number of tasks in odd-sets either decreases or remains unchanged. The inner loop ends if no additional tasks could be allocated.

2) Allowing higher communication costs for infeasible clusters. Global analysis does not take care of local communication. Tasks thus might not be allocated either because bandwidth of a local bus exceeds, or because local deadlines would be violated due to high communication delays.

If no further improvement is possible due to deadline re-synthesis, this backtracking loop modifies infeasible clusters according to allowed modifications on local buses. To this end, for each cluster with an odd-set imposing communication on the local bus backtracking calculates whether the local bus can be replaced by another one with higher bandwidth. This indeed raises the overall costs of the resulting system.

The ordering of applying modifications is determined by their costs. Backtracking maintains a separate list of possible modification on the local bus for each cluster. Iteration finishes if no further modifications are possible.
Note, that this backtracking loop influences complexity only linearly by the largest number of possible modifications among all local buses. After each modification, the inner loop is continued which leads to a further decrease of tasks remaining in odd-set, or no further improvements in the worst-case.

(3) Re-partitioning of odd-sets. The inner two loops finish with either all tasks allocated, or with odd-sets that cannot be further reduced. In the latter case, it is assumed that tasks of a given odd-set cannot be allocated to the respective cluster at all.

The outside backtracking loop thus takes the tasks of the odd-sets and assigns them to another cluster. To this end, global analysis is restarted, while tasks already allocated are assumed to be fixed now. In order to avoid re-allocation of tasks to previously assigned clusters (which would be useless) additional partitioning constraints are applied. These constraints have also effect on subsequent local analysis steps avoiding that already allocated tasks are put into odd-sets.

If no feasible solution is found by backtracking, the process could be restarted in a further loop with another initial partition for the global analysis, which largely determines the subspace the process is running on. Although such process will certainly find a feasible solution if any exists, this would probably lead to exceptional run times. Another option is to let the user allow higher costs. In this case, the analysis searches for solutions with additional modifications.

6.4 Experimental Results

The DSE algorithm with RTSAT-based local analysis has been applied to an example architecture consisting of a backbone FlexRay bus and three identical clusters (Figure 22a). There are four different ECU types available for modifications:

- Typ A: Costs=10 EUR, SpeedFactor=1.0
- Typ B: Costs=15 EUR, SpeedFactor=1.25
- Typ C: Costs=25 EUR, SpeedFactor=2.0
- Typ D: Costs=50 EUR, SpeedFactor=5.0

Each cluster initially consists of four ECUs of type A. The following modifications were allowed for each cluster:

- Any existing ECU might be replaced by any type,
- Two ECUs of any type may be added.

The task system consists of 79 tasks where 36 tasks are already deployed, producing the base load. That is, 12 tasks are deployed on each cluster (base loads are 3.8, 3.1 and 1.0). 43 new tasks are to be deployed on the system. The global backbone is a FlexRay bus with a static segment of length $L_{TDMA} = 60$ divided into 20 slots of length $\Lambda = 3$.

In the first iteration, global analysis proposed an initial distribution of tasks on clusters while assuring feasibility of the global bus. The allocation of messages to bus slots for the initial solution (Figure 22b, top) results in a bus utilization of 60%. In the next step, local analysis checked each cluster and found feasible allocations for clusters $S_2$ and $S_3$ satisfying costs predicted by the global analysis. For cluster $S_1$, an odd-set of two tasks $\{\tau^*_4, \tau^*_{19}\}$ was returned. The remaining tasks could be scheduled successfully assuming the predicted costs. In the second iteration, global analysis re-partitioned $\{\tau^*_4, \tau^*_{19}\}$ on cluster $S_2$, and predicted a cost increase of 10€. At the same time, two global messages became local, which reduced global bus utilization to 45%. The respective bus schedule is depicted at the bottom of Figure 22b. Resynthesis of deadlines where not applied to this example, and modifications of the local buses where not allowed. Local analysis of cluster $S_2$ with the new tasks and costs showed feasibility of the cluster, but found that modifications with costs of 5€ are sufficient.

The result is, that the process finds a feasible solution with additional costs of 55€. The overall runtime was $\approx 392$ s including three timeouts (each 120s) in the local analysis to search for solutions with less costs. The final deployment of base tasks and new tasks is depicted in Table 3 where “S1-P1” denotes ECU P1 of cluster $S_1$. The ECUs that were modified by the process are drawn bold in Figure 22a.
6.5 Termination and Complexity

It is quite easy to see that the discussed process always terminates. In each iteration, global analysis terminates because it is based on moving nodes from one partition to another where any node is moved to any partition at most once. This is a property of the underlying partitioning algorithm for which complexity has been discussed at Page 30. The complexity of the algorithm for finding feasible allocations of messages on the global bus has been discussed in Section 6.3.

Local analysis terminates because it is based on a SAT algorithm. Local analysis is the most critical part with respect to complexity, but also the one most hard to determine. We refer to the results of the experiment discussed in Section 6.4, and to the benchmark results of the original RTSAT algorithm in [36], since its complexity is comparable to the extended one.

Termination of the backtracking loop is also ensured. The inner backtracking loop re-synthesizes deadlines with respect to successfully allocated tasks. The loop is iterated as long as further tasks can be allocated due to refined deadlines. Complexity is thus bounded by the number of tasks and equals to $O(|T^*|)$.

The next outer loop iterates over all allowed modifications of the local buses. Since each modification is checked at most once and modifications apply in parallel for all clusters, complexity of this loop equals to $O(|\mathcal{M}_{lb}|)$ where $\mathcal{M}_{lb}$ is the set of allowed modifications with maximal size among all local buses. Because each iteration of the inner loop either further reduces the number of tasks among all odd-sets, or finishes, the overall complexity of both loops is $O(|T^*| + |\mathcal{M}_{lb}|)$.

Finally, after re-synthesis of deadlines and modifications on local buses, remaining odd-sets are re-assessed by global analysis in the outside loop. Global analysis maintains bookkeeping for task allocations, so that no task is allocated on the same cluster twice. The process stops if no more task allocations are possible, or if all odd-sets are empty. Thus termination is guaranteed, either with an empty odd-set or without a feasible allocation for all tasks. Because the number of tasks in odd-sets also never increases in the outside loop, we get overall complexity $O(|S| \cdot (|T^*| + |\mathcal{M}_{lb}|))$ where $S$ is the set of clusters.
7 Conclusions

A framework is presented for a design process that starts at high level specifications, and ends at implementations at the architecture level. In this paper, we use Matlab models as starting point. Function networks are employed as an expressive interface between such languages and the architectural level where real-time scheduling theory enables system analysis.

A carefully designed exploration process is proposed that finds satisficing solutions among a set of possible system architectures. To this end, a preliminary task creation partitions functionality into a more load balanced task network. The subsequent process iteratively explores the architecture space by a combination of an efficient global task partitioning algorithm and an also efficient SAT-modulo-(real time) theory approach. A novel extensible formalism to “sort” the design space into a more load balanced task network. The subsequent process iteratively explores the architectural level where real-time scheduling theory enables system analysis.

We prove termination of the exploration algorithm, and demonstrate, that the approach scales to realistic applications. While clearly an exact computation of optimal solutions in this complex design space exploration is infeasible, our approach yields satisficing solutions. Specifically, for given bounds on synchronization overhead between local and global communication designs, we provide solutions which meet given user-controlled cost constraints.

References


