

From Specification Models to Distributed Embedded Applications: A Holistic User-Guided Approach

Ingo Stierand, Günter Ehmen, Werner Damm and Detlef Janssen
University of Oldenburg

Matthias Büker, Stefan Henkler and Eike Thaden
OFFIS

ABSTRACT

We introduce a framework that aims at automating significant parts of the design flow in a typical scenario for embedded application development in the automotive domain. Given a specification model of a new automotive feature captured in Matlab-Simulink, the framework allocates new functions onto the devices of the hardware architecture such as ECUs and buses considering already deployed functions and the distributed nature of embedded systems used in the automotive industry. The framework is motivated by the iterative design process in industrial practice and subdivided into several steps. In the task creation process a balanced task structure is derived automatically from the specification model. Automatic code generation and execution time analysis for each task demands a semantics- preserving restructuring process of the Matlab-Simulink model. The task structure and the generated software tasks serve as input for the automated design space exploration process which has the goal to find a cost-optimized extension of the existing target hardware and an allocation of tasks on this modified target hardware. This allocation is sufficient to guarantee both system-level timing requirements and deadlines extracted from the Matlab-Simulink specification model. Engineers may guide the complete process by running it iteratively and tighten the constraints based on their expert knowledge. This semi-automatic user-driven and transparent optimization process helps to increase acceptance by engineers. For evaluation an industrial-motivated case study of a lane-change driver assistance system and an adaptive cruise control has been used.

CITATION: Stierand, I., Ehmen, G., Damm, W., Janssen, D. et al., "From Specification Models to Distributed Embedded Applications: A Holistic User-Guided Approach," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 6(1):2013, doi: 10.4271/2013-01-0432.

INTRODUCTION*

We introduce a framework that aims at automating significant parts of the design flow in the following typical scenario for embedded application development in the automotive domain: 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 time division multiplex (TDMA) based backbone 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 functions on electronic control units (ECUs), and either aim at 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. It finds a cost-optimized extension of the existing architecture, and an allocation of the tasks of the new functions such that all timing constraints are met.

We assume that the new feature 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

*This work was supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center SFB/TR 14 AVACS.

exploration process. The goal is to optimize the granularity of the task structure while maintaining causality constraints by balancing computational load and minimizing communication density. This is achieved by introducing a metric called *cohesion* which reduces task-switching times by avoiding too lightweight tasks and relieves the bus by keeping inter-task communication low. As an interface between this front-end process and the design space exploration process, we use function networks, which provide a formal semantic base expressive enough to represent different communication and execution paradigms and all timing related aspects [7]. The timing behavior of the tasks is obtained by existing worst-case execution time analysis tools [2, 33, 14].

In principle, the design space exploration problem can be formulated as an integer linear programming problem, where variables, encoding 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 [12] and ongoing investigations in using column- generation [1], 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 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 [28].

All together, our approach offers the following advantages: first, in contrast to research like [26] (see Section), our approach supports an 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 expressed in the AUTOSAR approach again and 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 test platform that allows to validate the results by executing (automatically) generated code on the platform in a simulation- in-the-loop environment.

The following section provides an overview of the framework discussed in this paper, and Section illustrates the case study that is used as a running example. The front-end process of translation from a given functional model and task creation is discussed in Section and. We formalize the design

space to be explored, and the optimization objectives in Section. The design space exploration process is discussed in Section. Section introduces the test platform together with actual test results for the given case study. Section provides an overview of related work, and Section concludes the paper.

DESIGN PROCESS OVERVIEW

The proposed design process is depicted in Figure 1, and starts with a Matlab-Simulink model that is translated into a function network (Step 1). This includes the translation of Simulink blocks into function nodes, and the derivation of worst-case execution times and node weights from C-Code generated from these blocks using existing code generators. To formally represent Matlab Simulink models we follow Lubliner et al. [25] where these models are defined as timed synchronous block diagrams (TBD). The *task creation* step (Step 2) merges functions to get more balanced systems while preserving semantics of the initial model.

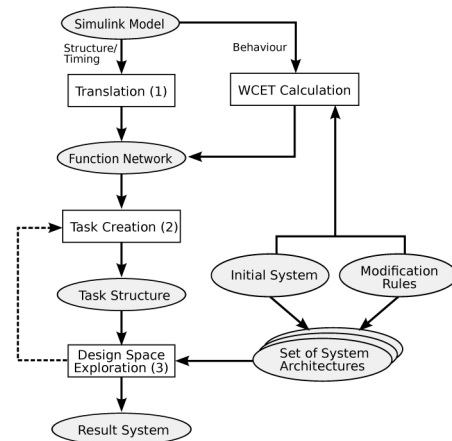


Figure 1. Process Flow

In Step 3 we start our design space exploration process. We have a three step approach which exploits the hierarchical structure of the target architecture: global approximate analysis, exact cluster-level analysis, and backtracking. In global approximate analysis, we use a formal analysis model for the global bus (in this paper restricted to be TDMA based bus), current load of cluster ECUs and backbone bus, and the task network to be allocated. A variant of the Kernighan/Lin algorithm partitions the task networks into subnetworks and allocates 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 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 “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 [27, 28] supporting architecture modifications.

A task network estimated to be feasible during global analysis can be refuted during the exact local analysis step due to the following approximations. To not lose 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. Finally, intra-cluster communication is not taken into account.

The backtracking step uses the local response time results of allocated tasks and the odd-set, and recomputes local deadlines taking into account exact response times for allocated tasks. Then it 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 this also fails, we re-partition the odd-set and iterate the process. The result is a solution that guarantees timeliness requirements and that 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.

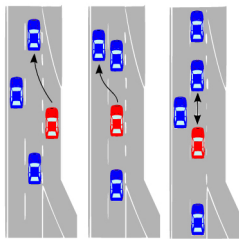


Figure 2. ViDAs features

THE VIDAS CASE STUDY

Implemented as a MATLAB/Simulink/Stateflow model, the *Virtual Driver Assistance* (ViDAs) case-study comprises *Adaptive Cruise Control* and *Lane Change Assistant* functionalities. By means of integrated vehicle sensors, the system is able to detect and to visualize traffic signs of speed limitations, which are installed at the roadside or above the driving lane. Further, an active ACC includes the automatic adherence to identified speed limits. At any time, the driver is capable of overriding or deactivating the system. The driving

simulation software SILAB of the *Wuerzburg Institute for Traffic Sciences GmbH*¹ enables the system's execution within a simulation environment.

The systems functionality is based upon state-of-the-art driver assistance systems established in the market. Technical details have been derived from the “Handbuch Fahrerassistenzsysteme” [34] and the standard ISO 15622 for ACC systems [19]. Further, design ideas of the ViDAs developers have influenced the implementation of the driver assistance system. Originally, the ViDAs system has been developed within a student project group as part of the computer science degree course at the *Carl von Ossietzky University Oldenburg*. Continuous enhancements of the system are further accomplished by research projects and student theses. Therefore, the ViDAs system corresponds to the research activities of the AVACS project as outlined in [35].

LCA - Lane Change Assistant

The primary task of the LCA is to assist the driver in changing the driving lane and in entering a highway (see left and middle part of Figure 2). Warning of vehicles in the “blind spot” area is also provided. In comparison to the formal LCA model of [35], the ViDAs LCA does not actively intervene in controlling the ViDAs equipped vehicle (ego-vehicle). By means of light signals installed in the exterior mirrors, the driver is informed by the LCA whether a lane change or entering a highway is currently possible without violating the safety gap to other traffic participants. In doing so, only the driving lane next to the ego-vehicle is regarded. A green light signal symbolizes a safe lane change. If a lane change is currently not possible due to the risk of negatively affecting the traffic safety or due to an identified restriction on lane changing, a red signal will be activated.

ACC - Adaptive Cruise Control

The ACC continuously maintains a safety gap to a preceding vehicle (see right part of Figure 2). Simultaneously, a constant velocity selected by the driver is observed as long as a preceding vehicle is not driving at a lower speed than the ego-vehicle. Furthermore, road signs are detected by the sensors of the driving assistance system so that an identified speed limitation may also override the desired velocity.

Operation and Display Elements

Activation of the ACC as well as selecting the desired velocity is provided by dedicated operation elements. A light signal represents the activation of the velocity option while the selected velocity is being indicated on a display. In summary, there is an operation element for adopting the current velocity of the ego-vehicle as well as three elements with preselected velocity values. These values are initially

¹Würzburger Institut für Verkehrswissenschaften GmbH, <http://www.wivw.de>

predefined but can be changed when the ego-vehicle has been stopped. The ACC function is deactivated whenever the driving speed drops below 30 km/h or in case of pressing the brake pedal.

The LCA is activated and deactivated by a dedicated operation element. A light signal next to the operation element outputs the element's status. An additional display indicates the currently valid velocity limitation or restriction on passing.

Limitations

In order to accomplish the realization of the ViDAS system as part of a student project group, the system environment have been constrained as outlined in the following:

Environmental conditions: The system shall only be used during daylight excluding unfavorable weather conditions such as rain or snow.

Roadway: The system is designed for two-lane highways as defined by the guidelines for constructing roadways [16]. Further, traffic signs have to be installed according to the instructions of [17].

Vehicles: The ego-vehicle has a maximum speed of 160 km/h whereas other vehicles drive at a maximum velocity of 200 km/h. It is assumed that all traffic participants adhere to the road traffic regulations.

Sensors: The vehicle's sensors are idealized and deliver realistic and thus no noisy data values at any time. Further, defective sensors are not regarded.

Additional information about the system is available at [3].

FROM SPECIFICATION MODELS TO FUNCTION NETWORKS

Function Networks

Like other task network formalisms, function networks [7] are directed graphs where nodes represent processing elements (tasks), and edges represent channels transmitting events between nodes. A *channel* may transport different events. Channel c_3 in Figure 3 for example transmits events from the set $\{d_1, \dots, d_n\}$.

Event sources allow to model events produced by the environment. In Figure 3, they are depicted as rectangles with filled circles. Source ϕ_3 represents for example a distance sensor delivering values $d_1, \dots, d_n$. The occurrence of events is defined in terms of event streams [30]. Common streams define for example periodic or sporadic event occurrences. Event streams for function networks are defined by a tuple $(\Sigma^{out}, P^-, P^+, J, O)$ where Σ^{out} is a set of events, $[P^-, P^+]$ defines an interval of a minimum and maximum period for event occurrences, J is the jitter, and O is an initial offset.

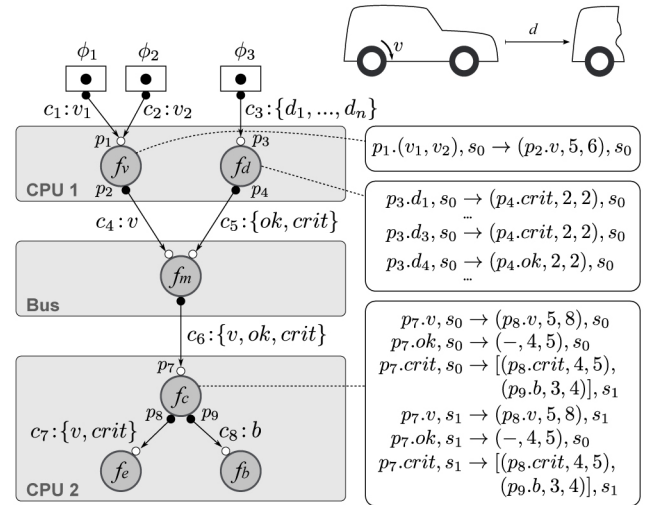


Figure 3. Adaptive Cruise Control System

The connection between nodes and channels is realized by ports. Activation of nodes is captured by their input ports (small white circles). An input port activates a node when at least one event has occurred at each incoming channel of that port. Node f_v in Figure 3 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. Combining multiple ports and multiple input channels allows modeling of complex node activations.

Function nodes employ internal state-transition systems to model for example functions that are sensitive to incoming events, and data access to, e.g. shared variables and FIFO buffers. Node f_c for example sends a braking event b whenever it is activated and the last captured distance was critical (*crit*). 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 ms to be sent to port p_8 .

To simplify modeling data flow, the function network formalism is extended by data nodes, that are special function nodes modeling explicit data storage. We define different types of data nodes as persistent ones like *Shared* variables and *FIFO* buffers, and volatile ones like *Signals*. Another data node type is the *Finite Source* (*FSource*) producing an initial event at its output port at system startup while emitting the next event not before an event was received at an input port. This node type is used to model cycles in function networks. Its semantics is very similar to pre-allocated events in task networks with cyclic dependencies [21]. A further enrichment is the introduction of a new channel type named *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 function node. Read channels are depicted as dotted arcs.

A function network is defined as a tuple $FN = (\Sigma, \mathcal{P}, \mathcal{C}, \Phi, \mathcal{F}, \mathcal{D})$ where:

- Σ is a finite alphabet. Events are tuples $e = (\sigma_1, \dots, \sigma_k)$ with $\sigma_i \in \Sigma$. $\dot{\Sigma} \subset \Sigma^*$ is a finite set of events.
- $\mathcal{P} = \mathcal{P}^I \cup \mathcal{P}^O$ is a set of input and output ports,
- $\mathcal{C} = \mathcal{C}^A \cup \mathcal{C}^R \subseteq (\mathcal{P} \times \mathbb{N}^+ \times \mathbb{N}^+ \times \mathcal{P})$ is a set of channels $c = (p^{out}, \delta, p^{in})$ where $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ is a delay interval, $c \in \mathcal{C}^A$ are *activation* channels and $c \in \mathcal{C}^R$ are *read* channels leading exclusively from data to function nodes.
- Φ is a set of sources $\phi = (EP, P^{out})$ where $EP = (\Sigma^{out}, P^-, P^+, J, O)$ is an event pattern, $\Sigma^{out} \subseteq \dot{\Sigma}$ are the events transmitted by the source. $P^{out} \subseteq \mathcal{P}^O$ is a set of output ports.
- $\mathcal{F}$ is a set of function nodes $f = (P^{in}, \mathcal{A}, P^{out})$ where $P^{in} \subseteq \mathcal{P}^I$ is a set of input ports, and $P^{out} \subseteq \mathcal{P}^O$ is a set of output ports. $\mathcal{A} = (S, s_0, T)$ 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

$$T : P^{in} \times \dot{\Sigma} \times S \rightarrow ((P^{out} \times \dot{\Sigma}) \cup \{\perp\} \rightarrow \mathbb{N}^+ \times \mathbb{N}^+) \times S$$

mapping combinations of ports, incoming events and states to ports, delay intervals and successor states. Symbol $\perp$ denotes “no output port”.

- $\mathcal{D}$ is a set of data nodes $d = (P^{in}, \delta, b, P^{out})$ where $P^{in} \subseteq \mathcal{P}^I$ is a set of input ports, $\delta \in (\mathbb{N}^+ \times \mathbb{N}^+)$ is a delay interval, $b \in \{FIFO, Shared, Signal, FSource\}$ is a data node type. $P^{out} \subseteq \mathcal{P}^O$ is a set of output ports. $\diamond$

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 4). In the following we consider only end-to-end deadlines for simple linear task chains. More complex chains are divided into sets of linear chains, which is always possible. The corresponding deadline spans the time between the activation of the first node and finishing the execution of the last node of the chain. In our setting, all deadlines are derived automatically from the Matlab-Simulink model.

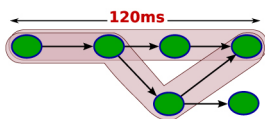


Figure 4. End-to-end Deadline

Translating Simulink

In our design process, a function network is obtained from a Matlab Simulink model by a set of translation 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 *synchronous sets*, which are connected blocks with the same sample time [10], the Matlab Simulink concept of *Rate Transition Blocks* is mapped to function networks. Timing information is obtained by deriving event patterns for specific nodes in the function network. The behavior of the specification is translated into executable code by employing (existing) code generators such as Realtime Workshop or TargetLink. Worst case execution times (WCETs) are calculated for the resulting code blocks to assign weights to the respective function nodes. For this, we use the static program analysis tool aiT [14]. For translation, we pick up the basic idea of [25] where Matlab Simulink models are defined as *Timed Synchronous Block Diagrams* (TBDs). The simulation steps where a block is executed are determined by its sample time, which consists of a *period* and an initial phase offset *init*. The base rate of a Simulink model is defined as the greatest common divisor of all periods and initial offsets that occur in any sample time. The TBD is flattened as described by Lubliner [25] while the hierarchy of subsystems is used to generate constraints for the task creation (see Section). As in [25], we assume TBDs to be acyclic in that all cycles must contain at least one Moore-sequential block such as a “Unit Delay” block. In the following we shortly introduce the translation concept.

First, we define for the function network translation a single event source ϕ_{br} , which is running with the base rate br as period. To model the different sample times, special function nodes called *period multiplier* are defined, which are triggered by events from ϕ_{br} and convert the base rate to the period of the respective sample time. This assures the synchronization of the initial events of all synchronous sets. A period multiplier is defined as a function node with a transition system that multiplies the period P at its input port with a factor k while adding an offset off . This offset determines the number of periods until the first event is emitted. In Figure 5 a simple Simulink model is depicted with its function network translation beneath. An example for a period multiplier is the function node denoted as ST_1 , which converts the base rate into the sample time ST_1 .

Each *ordinary* block is translated to a function node with one input port, and all incoming signals of a block are translated to channels that are synchronized at this input port. The transition function consists of a single state and one transition that is fired as soon as all input events have arrived. This transition creates all output events and its delay is determined as the worst case execution time of this block. An

example for the translation of an ordinary block is block *Add₁* in Figure 5. It has two incoming signals, which are translated to channels synchronized at a single input port of the translated function node.

Each *rate transition* block with an input signal from block *a* and an output signal to block *b* is translated to a function node that converts the sample time ST_a of block *a* to the sample time ST_b of block *b*. We assume that the respective periods $period_a$ and $period_b$ are integer multiples i.e., either $period_a = k * period_b$ or $period_b = k * period_a$ ($k \in \mathbb{N}^+$) and $init_a = init_b$.

If $period_a > period_b$, block *b* requires its input data more often than it is produced by block *a*. Thus, the rate transition block stores the latest value 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 a set of period multiplier nodes triggering the rate transition block whenever it is not triggered by the preceding node but an activation is required by the successor node due to its sample time. The set of period multiplier nodes produces events with an event pattern implementing $ST_a \setminus ST_b$, which results in the set of sample times that is needed to compensate the mismatch between ST_a and ST_b . In Figure 5, the block *RTB* represents a rate transition that leads from sample time $ST_1 = [6,0]$ to sample time $ST_2 = [2,0]$. In the function network representation, this is realized by function node *RTB* with channels from two period multiplier nodes, one for each sample time in $ST_2 \setminus ST_1 = \{[6, 2], [6, 4]\}$.

If otherwise $period_a < period_b$, *b* must only be activated every k^{th} time where $k = \frac{period_b}{period_a}$. This is realized in the function network by creating a period multiplier node transforming $period_a$ to $period_b$ by applying the factor *k*. We assume that rate transition blocks always have exactly one input and one output signal. 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.

For both translation and the following task creation, preserving semantics of the original specification is a key issue. In particular, we have to take care that ordering of block executions is maintained and that for each execution the currently available inputs match those of the original semantics. Additionally, we have to ensure that all functions of a simulation step are executed before the next simulation step starts. Thus, we define end-to-end deadlines for each chain of nodes that may be executed in the same simulation step. The length of the deadline is determined by the base rate *br* of the Simulink model. More details on preserving semantics can be found in [6].

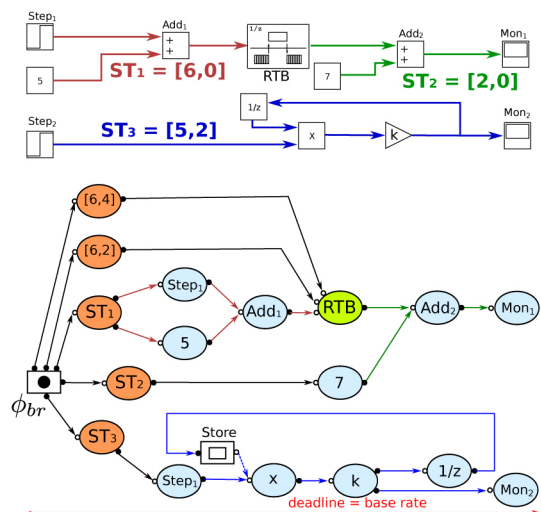


Figure 5. Translating Simulink Model to Function Network

TASK CREATION - SEMANTICS AND APPLICATION

A function network that results from translation of a Simulink model is typically unbalanced, consisting of a large number of nodes with high variance in computational node weights, which estimate the load a node potentially produces on an ECU. If we treat each node as a single task, this would result in a large communication overhead when many lightweight tasks are spread over the distributed hardware resources in the pursuing design exploration process. Accordingly, we want to obtain a more suitable task set by merging function nodes into tasks.

The proceeding of Simulink Embedded Coder is to put all blocks with the same sample time into one task. These may not only be blocks of the same synchronous set but also of independent sets sharing the same sample time which may lead to very “heavy” tasks. For the execution on a distributed real-time system, this strategy precludes any of these blocks from being executed concurrently, which increases the risk of deadline violations. Nevertheless, nodes of the same synchronous set are still good candidates to be executed in the same task [10]. But due to the possibly high variety of the number and weights of nodes in synchronous sets not all sets would necessarily result in useful tasks. For example, we do not want to allow arbitrary large weights because those tasks 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, which leads to thrashing. From the perspective of the design space exploration, 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