Christian-Albrechts-Universität zu Kiel

Diploma Thesis

Model-Based Design of Distributed Time-Triggered Architectures

An Avionics Case Study

Hauke Fuhrmann

April 15, 2005

Department of Computer Science
Real-Time and Embedded Systems Group

Advised by:
Dipl.-Inf. Jan Lukoschus,
Christian-Albrechts-Universität zu Kiel
Dipl.-Ing. Jörn Rennhack, Airbus Deutschland GmbH

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

_____

Abstract

Safety-critical applications, *e.g.*, in automotive and aerospace industry, demand highest dependability. Such systems are often distributed hard real-time systems with severe system response time constraints. To cope with these requirements and the need of high reliability, the time-triggered architecture provides a deterministic and fault-tolerant execution and communication scheme, which renders it suitable for these applications.

The model-based system design assists the developers in handling the high complexity of serious distributed real-times systems in general as well as the details of the time-triggered architecture in particular.

This diploma thesis describes the model-based system design of a time-triggered architecture and lists the process in some detail regarding the dichotomy of global and local specifications and their conflation in a modelling suite. It describes the modelling of a safety-critical application: a high-lift flap system which gets implemented in a time-triggered architecture. This thesis lists what has been done so far and what is planned for the future in this context.

The practical application of the modelling tools raised the question on response time requirements. How can modelling tools handle end-to-end latency and how do they do it nowadays? Do they do all that is possible? Is it not possible to handle it after all?

This diploma thesis tackles this question by analysing the origin of latency in real-time systems in general and in the TTA in particular, where it depicts worst-case latency. It analyses the workflow of the current modelling tools with respect to the response time requirements. It is discussed which properties are critical to the end-to-end latency in a TTA and how the settings of the modelling tools influence these properties.

The thesis shows that both, validation of the system response requirements and the synthesis of the system configuration from the response time requirements is not possible with the current toolchain. Additionally it introduces an approach in which the conflation of the fundamental specification dichotomy – global vs. local – in a modelling suite could help to get a convenient response time analysis function after all and therefore the modelling process could be enriched and thereby save a lot of testing efforts.

Keywords Model-based System Design, Time-Triggered Architecture, Response Time Requirements

Contents

List of Figures

1 Introduction

Reducing costs while increasing both features and safety is the ambition of modern transport industry. In order to stay competitive, manufacturers of automotive, aerospace and industrial products increase the feature palette of their products. With advances in the development of microcomputers, these entered as embedded systems cars and planes. Electronic systems are flexible and relatively low-priced, so they add new features to the products or substitute inflexible and non modular mechanical systems in order to increase flexibility and safety and maybe to reduce production costs a piece.

With the increasing growth of electronic components in vehicles, the need of communication between these components rises. Distributed real-time systems result and their advances push them into safety critical fields of application where up to now mechanical systems guarantee dependability. Omitting even mechanical backup systems which hitherto caught faults in electronic systems leads to the so called X-by-wire systems. They promise flexibility by decreasing both space and weight requirements on the one hand and reducing costs after the development process by economies of scale on the other hand, as the unit costs of mechanical systems exceed electronic systems by far. Modularity of electronic systems introduces a better reusability and higher flexibility by defining hardware and software interfaces that allow a wider variety of possible implementations. Reusability and the better choice of implementation techniques can lead to savings in future development efforts and can ultimately result in increased dependability.

With the growing complexity of distributed systems the intercommunication within and between subsystems is increasing. The standard event-triggered communication buses such as CAN [ISO93] show no deterministic behaviour. Therefore much effort is undertaken to keep the systems safe. In today's cars, the multitude of event-triggered communication buses, which may be redundantly implemented, lead to a complex tangle of wiring, while dependability can hardly be ensured. In contrast, the time-triggered communication approach in the *time-triggered architecture* (*TTA*) [KB03] offers deterministic, fault-tolerant communication services with features that enable the developers to better manage complexity and to find design flaws earlier in the development process. Thus the seemingly higher development effort might pay off in the end. The *Time-Triggered Protocol* (*TTP*) [KG92] is an example of the

communication subsystem in the TTA.

Another advancing key technology is the *model-based system design*. Modelling suites such as Matlab/Simulink/Stateflow, SCADE or Mission Level Designer give us the chance to model a system prior to its physical implementation. This helps to find errors in the development process in early stages: Design faults in the specification can be revealed or the whole system can be simulated although some of the components do not yet exist physically. This might help to find errors as soon as possible and can save effort, time and money because in later design phases the costs of eliminating errors are likely to be much higher.

For best performance in the industrial development process the two technologies, TTA and model-based system development, should be combined. Thus the development process leads to a model-based system design of the TTA. First tools exist to merge modelling and TTA.

Practical use with the tools raises the question of usability. Do the tools of today offer all the features they could? The system models have to be fed with a lot of information that is specific to the communication subsystem and that the developer has to devise. Do not some of these properties arise from the general system requirements? This work tackles this question and shows how the tools could help the developer better, to set the mandatory system properties correctly.

This question is motivated by an accompanying research project where we model a TTA system. The Real-Time/Embedded System Group of the Christian-Albrechts University of Kiel is partner in the European IST Research project *DECOS* (*Dependable Embedded Components and Systems*) [DEC]. The project should deploy a process of developing embedded systems that are built on COTS (commercial-of-the-shelf) hardware and software components and nevertheless enable us to develop safe systems. The major objective is to research the compositional system framework and to develop a set of generic hardware and software components usable on various platforms including the *time-triggered architecture*. The key to the problem is to develop fundamental and enabling technologies which are independent of domain and technology in order to facilitate the paradigm shift from *federated* to *integrated* design of dependable real-time embedded systems. This shall lead to reduced *development*, *validation* and *maintenance* costs in both the software and the hardware domain.

For demonstration of the research and development results, several test benches will be implemented employing the new technology. One of these test benches is a high-lift flap system for aircrafts. As such a system is highly safety-critical, its safety features were implemented by mechanical parts up to now. The most serious drawback of this solution is the inflexibility and the

high weight of the system. Implementing these safety relevant functionalities by electronic components with the DECOS methodology and tools can increase safety, modularity and even possibly decrease the weight of the system. The test bench will be physically implemented by the DECOS Project partners. It demonstrates the DECOS technology in this application.

Our task is to develop a model of this high-lift flap test bench including the logical behaviour of the system. The model will be built in the *SCADE Suite* by *Esterel Technologies*.

The main attention is drawn to modelling the time-triggered communication between the single components. Thus a TTA coupling to SCADE is needed and provided by Esterel Technologies and *TTTech*.

1.1 Outline

As this work deals with the key technology of time-triggered communication and its notions will be used through the whole work, the time-triggered architecture will be introduced in some detail in Section 1.2.

The second key technology of this work is model-based system design. This topic will be introduced in 1.3 and includes a description and comparison of the modeling suites of *Mathworks* with the *Matlab* family and *Esterel Technologies* with its *SCADE Suite*. Both are used in the project.

The DECOS project is the context of this work and therefore the project will be pictured in Section 1.4 and especially the details of the aerospace demonstrator will be given in Section 1.4.1.

The work we have done so far and the results of the DECOS project is described in Chapter 2, where Section 2.5 outlines the future work.

The question whether and how the tool chain for the TTA can be enriched by the automation of system requirement realization is tackled in Chapter 3.

1.2 The Time-Triggered Architecture

The *time-triggered architecture* (*TTA*) [KB03] is a distributed, synchronous, fault-tolerant architecture, which tackles the needs of safety-critical real-time systems.

The *TTA* establishes a frame for data processing in the area of distributed embedded real-time systems with highly reliable applications. It sets up the computing infrastructure for the implementation of applications and provides mechanisms and guide lines to partition large applications into nearly au-

tonomous subsystems along small and well-defined interfaces so that the complexity of the evolving product can be controlled.

1.2.1 TTA and the communication subsystem

The TTA comprises distributed host computers that intercommunicate via connected communication controllers. In a TTA, both, the task execution at the host computers and the communication actions are *time-triggered*.

> *A large real-time application is decomposed into nearly autonomous clusters and nodes and a fault-tolerant global time base of known precision is generated at every node. In the TTA this global time is used to precisely specify the interfaces among the nodes, to simplify the communication and agreement protocols, to perform prompt error detection, and to guarantee the timeliness of real-time applications.* [KB03]

As TTA is a general notion, there are different implementations available. The *Time-Triggered Protocol* (*TTP*) [TTA03] is a communication protocol for the TTA and specifies its communication subsystem. Other approaches are forthcoming. *Byteflight* [BPG00], *FlexRay* [BMH+01] and *TTCAN* [FMD+00] are worth mentioning for example. This diploma thesis, however, examines the model-based system design of a TTA using the TTP as the communication subsystem. Therefore the technology will be explained for TTP in particular. A short introduction and comparison of *byteflight*, *FlexRay* and *TTCAN* is given at the end of this section in Section 1.2.7 on page 11 after the technical introduction of TTP.

The TTA with TTP as communication subsystem consequently follows the *single fault hypothesis*:

> Any electric/electronic component in the system can fail (*e.g.*, a cable a connector, a whole ECU, ...), but no two independent faults will occur within a certain amount of time. This amount of time is sufficient to fully recover from the fault, *e.g.*, by repair or reconfiguration.

Following the single fault hypothesis implies that the whole architecture is designed in such a way that no arbitrary single fault can affect the safety of the whole system. As faults cannot be prevented in general, it must be ensured that they only affect a predefined *fault containment region* , which is separated from the rest of the system. Special techniques and algorithms must ensure

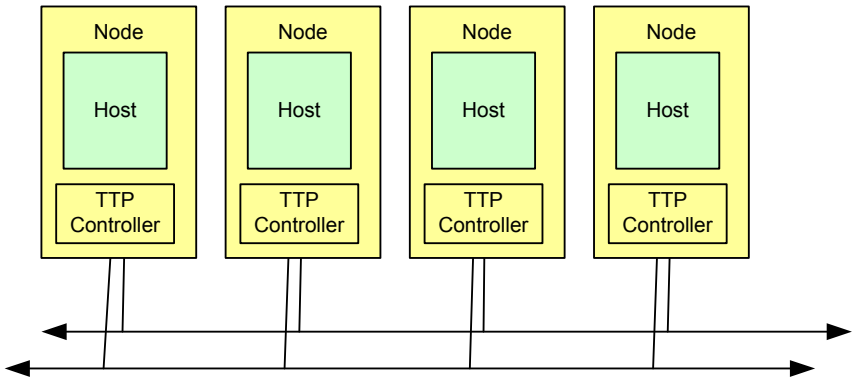


Figure 1.1: Nodes of a TTP cluster: Connected via a redundant bus and comprise a host computer and an independent TTP controller

this as well as methodologies such as *redundancy* must buffer the loss of single components.

In the language of TTP the distributed ECUs are called *nodes*. Each node comprises a host computer that executes some application and a network controller that implements the *Time-Triggered Protocol* (*TTP*) [TTA03].

1.2.2 The Time-Triggered Protocol

The *Time-Triggered Protocol* (*TTP*) describes the communication scheme of the system. The set of nodes participating in communication via TTP is called the *TTP cluster*. Usually a TTP cluster is connected by a bus. To meet the single fault hypothesis, the bus has two independent channels. Messages can be transferred on either channel, but safety critical messages must be transferred on both channels simultaneously to tolerate a single fault on one of the channels. This topology is depicted in Figure 1.1.

The host and the TTP controller within a node are independent of each other. This guarantees that a fault is kept in one part and thus prevents fault propagation to the other. The TTP controller manages the sending of messages onto the TTP bus and the receiving of messages from it. The host must process the incoming messages and produce the outgoing ones. The interface between host and controller is called Communication Network Interface (CNI) and it is implemented as a RAM Section where both partners have dedicated reading and writing access (see Figure 1.2). Host and TTP controller may not be perfectly independent after all, because the TTP controller freezes if the host application does not regularly give life signs.

The access to the bus follows the *time division multiple access* (TDMA) scheme. In an event-triggered communication system such as Ethernet, every

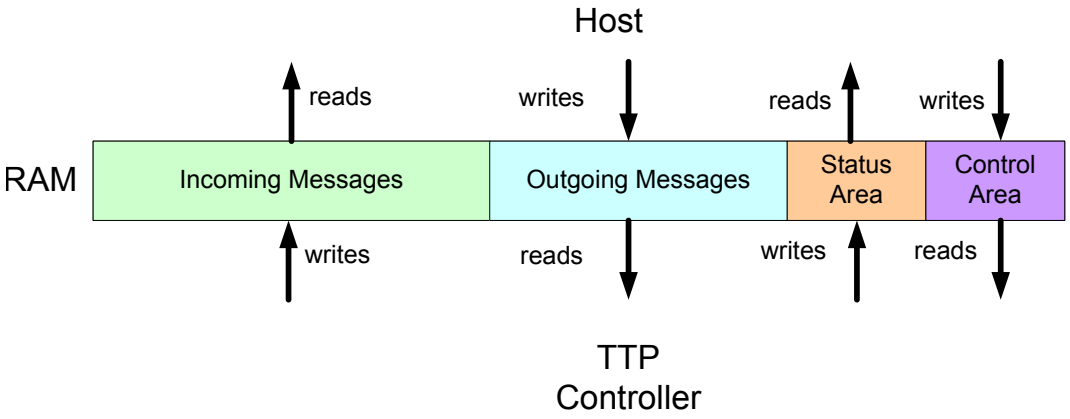


Figure 1.2: The Communication Network Interface for data exchange between host computer and TTP controller within a TTP node

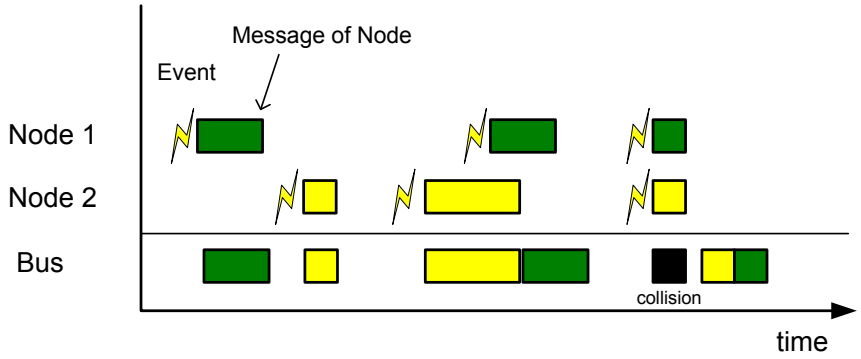


Figure 1.3: Problems in an event-triggered communication system: No deterministic latency and shrinking bandwidth when traffic is rising by the increase of collisions

node tries to send its message whenever it likes to, *i.e.*, when some special event makes the sending of a message necessary. This leads to the problem depicted in Figure 1.3, which is the introduction of *jitter*, *i.e.*, no predictable message latencies. As the name says, the messages in TTP are sent at special points in *time*.

Every active node may get exactly one *time slot* at which his TTP controller sends his message. The queue of these slots is called a *TDMA round*. Nodes do not need to have a time slot in every TDMA round: So called *multiplexed nodes* can send their messages in a multiple period of the TDMA round period and therefore get a time slot only every second TDMA round for example. The set of TDMA rounds that constantly repeats is called the *cluster cycle* (see Figure 1.4).

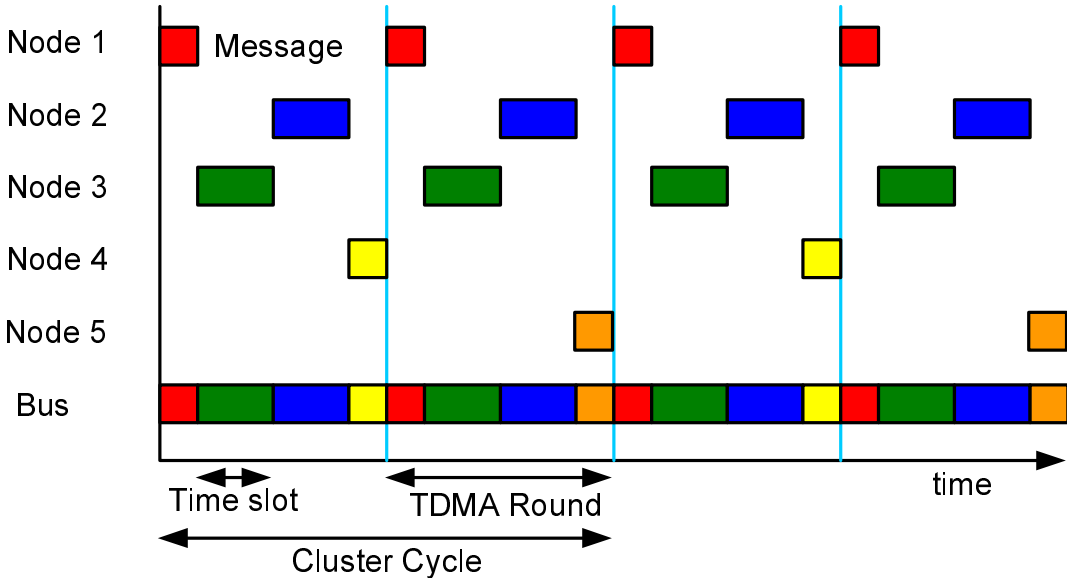


Figure 1.4: The TDMA scheme of a TTP network

So every node regularly sends his messages at the same period of the TDMA round length.

The schedule that controls the communication behaviour is called *Message Descriptor List (MeDL)*. Every TTP controller must have its own copy of the MeDL to know the timing information of the TTP bus.

1.2.3 Underlying Services

The TDMA access scheme demands a clock synchronization algorithm. This is implemented as a low-level service in the TTP controller and synchronizes their clocks and propagates the synchronized time to the host computers. It uses a fault-tolerant clock synchronization algorithm that tolerates one Byzantine fault at a minimum of four active nodes in a cluster.

Every message includes status information about the status of the sending node. That especially includes a *membership vector* that indicates which nodes within a cluster are still available and which ones are considered to be disabled due to the loss of messages or because they do not react any more. An *implicit acknowledgement* algorithm uses this information to determine if the sent message of this node was correctly submitted or if a disturbance prevented the transmission.

The *membership algorithm* uses this information to check which nodes agree upon the status of the whole system, *i.e.*, the nodes that correctly participate

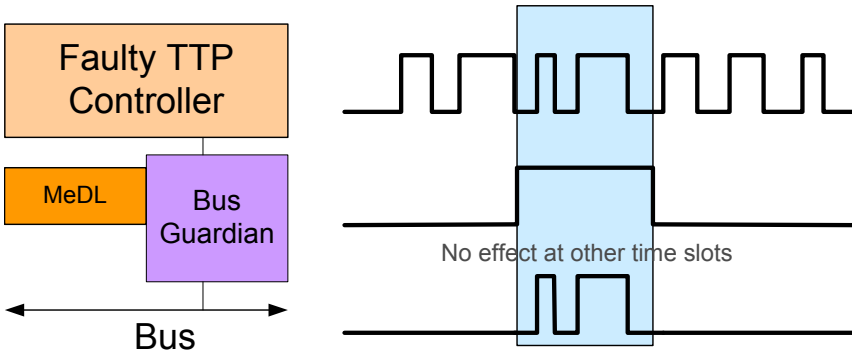


Figure 1.5: A bus guardian prevents a babbling idiot to block all traffic at the bus

in the communication. A *fault silence* strategy forces each disagreeing node to enter the *freeze state* and stop communicating. This ensures that only correctly communicating nodes are active.

The membership algorithm is augmented to a *clique avoidance* algorithm that avoids the constitution of several groups of nodes which agree with each other but not with the different other groups. The algorithm ensures that there is only one active clique in a TTP and that it comprises at least half of the nodes.

For saving bandwidth the status information is not explicitly sent with every message but might be implicitly encoded in a *Cyclic Redundancy Check (CRC)* which any message includes. Thus the receiver might not see *how* its status differs of the sender but *that* it differs.

The *babbling idiot* is a faulty node that accesses the bus at points in time where he is not allowed to send. This behaviour cannot be prevented in an event-triggered network. An independent *bus guardian* tackles this problem in the TTP. It is an independent unit that is located between the TTP controller and the TTP bus. The MeDL is separately stored at the guardian and it therefore knows when the TTP controller is allowed to send data. Hence it allows only such signals to pass that are coming in the right time slots (see Figure 1.5). A bus guardian prevents a node from keeping other nodes from sending. An upgraded guardian could even refresh the physical signals of a TTP controller in order to avoid so called *slightly off specification (SOS)* errors. But it cannot prevent faults in the value domain, *i.e.*, prevent a node of sending wrong values.

1.2.4 Bus topologies

By default a TTP cluster has a bus topology as seen in Figure 1.1. This successively connects the nodes with the bus and therefore only one physical cable is needed (respectively two cables if the two channels are spatially separated). Hence this topology is good for the use in widely distributed applications where cables must be saved for weight, space and complexity reasons. Unfortunately this topology introduces fault modes that cannot be prevented: *Spatial proximity faults*[1] can affect the whole network, if the fault containment regions reach more than one node. The TTP bus inherently constitutes a fault containment region. If a single event cuts the cable somewhere or transient disturbance compromises the bus, communication of more than one node might be affected. And as the two channels merge in every node they cannot be fully spatially separated and therefore the fault mode cannot be prevented completely.

The alternative is the *star topology* shown in Figure 1.6. Two independent central *star couplers* are connected to each node and broadcast the incoming signals to all other nodes. Local bus guardians at the nodes are omitted. Each star coupler has one bus guardian that broadcasts messages only if sent in the right time slot. This protects the system from spatial proximity faults and does not broadcast signals that are slightly off specification if the global star couplers are equipped with signal reshaping functionality. Therefore this topology is to be preferred in safety-critical application domains.

The drawback is obvious: The central star couplers require cabling from each node to both of the couplers. This does not makes the star topology applicable for several application domains.

1.2.5 Fault-tolerant communication

The nodes in a TTP network behave *fail-silent*. That means whenever a TTP controller detects divergent behaviour, it enters the freeze state and stops sending messages to the bus. Thus the system can lose a source of important messages such as sensor values if a fault occurs at the corresponding node. Therefore the *fault-tolerant communication layer* allows to specify more than one node for the implementation of a subsystem. This is shown in Figure 1.7: Two or more nodes are grouped together in *fault-tolerant units* (*FTU*) that all implement the same subsystem. Hence this subsystem sends its messages redundantly: Every node of the FTU sends the message and therefore the

[1] A common fault where a single event affects more than one part of a system due to the spatial proximity of the system parts. This could introduce the violation of the single fault hypothesis.

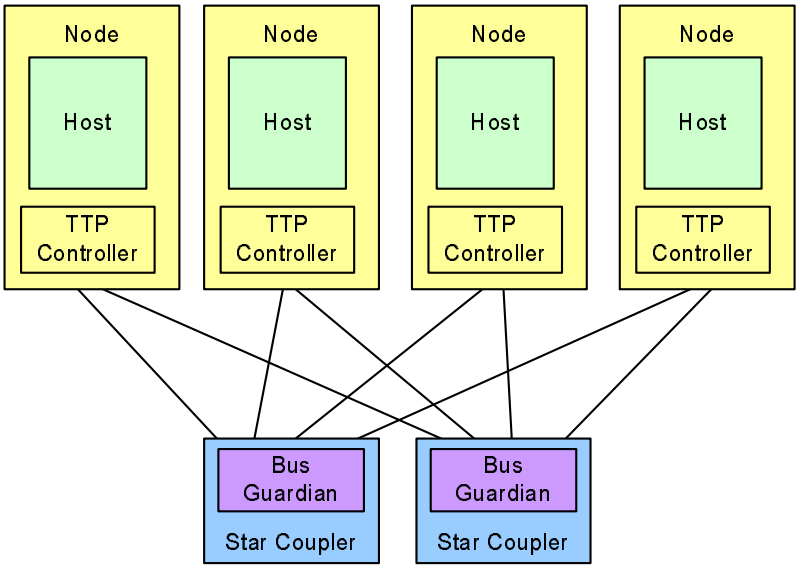


Figure 1.6: The star topology for TTP networks

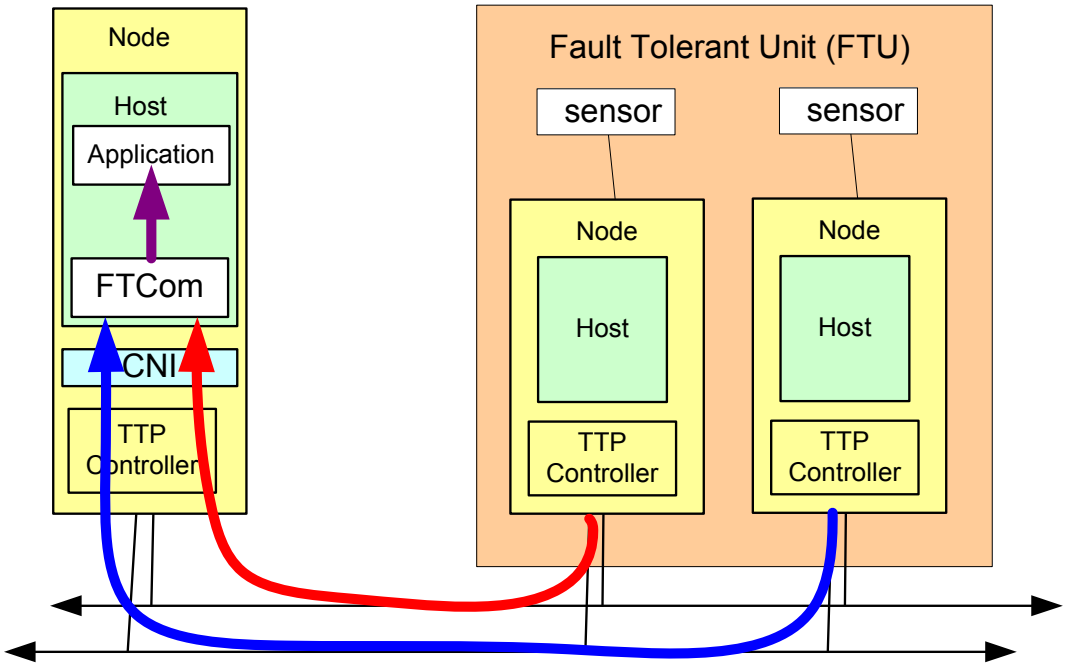


Figure 1.7: Principle of the FTCOM layer

receiver gets multiple versions of it within one TDMA round. The developer of the receiver shall not be charged with additional information he has to handle. Therefore a little software layer in the receiver's host application— the *fault-tolerant communication layer* (*FTCOM*) —fetches all the messages of the FTU nodes, merges them and passes only one message to the receiving application. As sensor values of different sensors can slightly differ, the merging of the values uses a dedicated algorithm like an average value or a voting between more than two values. The algorithm has to be chosen resp. implemented by the developer. If one of the FTU nodes fails, the FTCOM layer uses the remaining messages of the FTU to pass the value to the application.

The fault-tolerant communication layer is not part of the TTP specification. The time-triggered approach enables the developer to use this way of redundancy, but the FTCOM functionality must be implemented by the developer in the host application. In order to decouple application development from low-level dependability considerations, tools could and already can (see Section 1.3.2.1 on page 20) generate the FTCOM layer automatically. An implementation of the FTCOM layer in dedicated hardware could dispense the host from non-negligible runtime- and memory-consumptions. This will be done in the DECOS project (see Section 1.4 on page 34).

1.2.6 From TTP to TTA

TTP only describes the communication scheme of the TTP cluster. As the host computer of each node is completely independent of the TTP controller, it can fetch the messages at any arbitrary time. It can process its tasks whenever it wants and put messages to the CNI to be sent at any time as well (whereas a mutual exclusion mechanism has to prevent access to the data while they are inconsistent, *e.g.*, at the moment the TTP controller writes the data onto the CNI).

The time-triggered architecture (TTA) extends the time-triggered scheme from the network to the host application. In a TTA the tasks are executed with a time-triggered operating system [OSE01] and another schedule at each node (the *Task Descriptor List* (*TaDL*)) controls the task execution. As the MeDL is a global schedule, the scope of a TaDL is only one node. It is generated according to the global MeDL in order to run a task that uses a message, after the message has been received, and finishes the task before a result has to be sent. Figure 1.8 shows an example. The host computer can use the synchronized time that the TTP controller provides and therefore the scheduling of the tasks at the right times is possible.

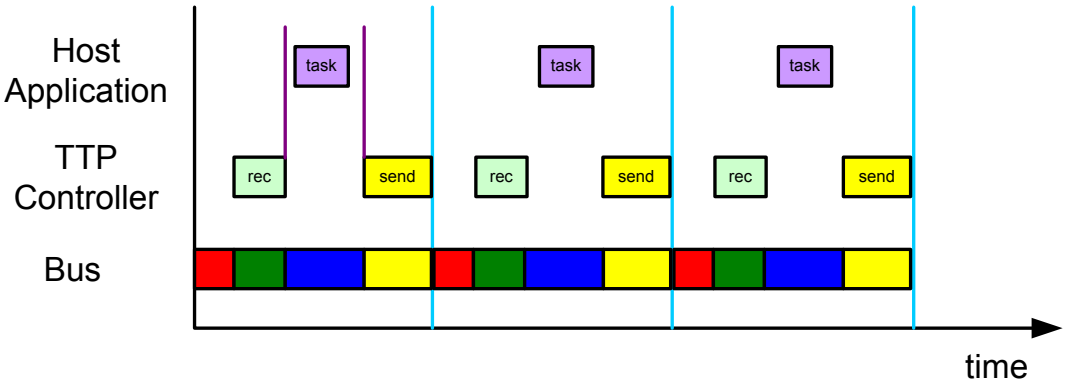


Figure 1.8: The time-triggered architecture extends the timing constraints to the host computer

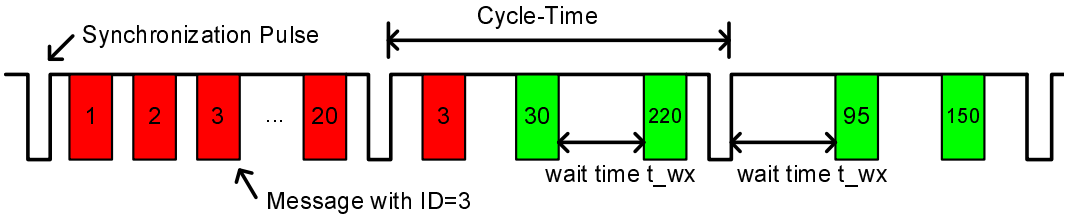


Figure 1.9: Flexible Time Division Multiple Access in *byteflight*

1.2.7 Alternative communication subsystems

byteflight Byteflight [BPG00], developed by BMW, uses a technique called *flexible time division multiple access*. The bus access is divided into time cycles of a static period where a synchronization pulse of a master node indicates a new cycle (see Figure 1.9). Each node resp. message type gets a unique identifier. Each message with a certain identifier may be sent once in a cycle but does not have to be sent. Due to individual static wait times t_wx corresponding to the identifiers, lower identifiers get preferred access to the bus each time cycle. Therefore critical messages with low identifiers will likely be able to send while messages with high identifiers might have to wait until a cycle has less traffic of higher priority messages. Furthermore, access to the bus for a few high-priority messages (coloured red) can be guaranteed by static analysis of the system during design time. Therefore the bus can be used synchronously. Other messages (coloured green) use remaining bandwidth for asynchronous access. This rather event-triggered approach comes very close to a time-triggered communication scheme.

FlexRay *FlexRay* [BMH+01] is being developed by a consortium of BMW, DaimlerChrysler, Motorola and Philips. Its operation is divided between time-triggered and event-triggered activities. While the general communication is strictly time-triggered, the TDMA-round is partitioned into a static time-triggered part and a dynamic event-triggered part. The division is set at design time. In the event-triggered part the nodes communicate using the *byteflight* protocol.

As FlexRay is still under development, Rushby [Rus01] states that many problems have not yet been solved or have not been as consequently tackled as in TTP. For example FlexRay does not provide interactively consistent message transmission. It provides no membership or failure notification service and contains no mechanisms to control *slightly off specification*[2] faults.

Nevertheless, the mixing of time- and event-triggered operation makes FlexRay interesting. It is potentially important because of the industrial influence of its developers.

TTCAN The *Time-Triggered CAN* (*TTCAN*) [FMD+00] protocol is realized in software in a higher layer on top of the event-triggered CAN [ISO93] protocol. A time master sends reference messages that indicate a new basic time cycle and to which the other nodes can synchronize their clocks. Within a basic cycle the time is divided into time slots. A slot can be assigned to a special message for purely time-triggered communication. Additionally, slots can be used as arbitrating windows, for which the nodes compete for bus access just as in the regular CAN communication.

For some applications it might be necessary to synchronize the cycles to events, *e.g.*, synchronize communication and motor rotation. For these cases the periodic transmission of messages can be discontinued until the event has taken place.

Unfortunately, TTCAN has no systematic fault tolerance, no a priori redundant channels and its transmission speed is quite slow due to the underlying CAN protocol. The features such as the synchronization of communication to external events show that it was designed for the automotive industry, where CAN communication is widely used and the same hardware can be used for TTCAN. However, it cannot be used for safety-critical applications.

[2] If an electric signal of one controller is slightly not within the specified valid ranges, some other controllers might accept the signal as correct and others might reject them as invalid. This could lead to inconsistent system states.

1.3 Model-based system design

The complexity of the systems obviously rises by introducing more and more electronic systems which intercommunicate. The system developer must handle topics of safety, reliability, maintainability, availability, security and reusability.

One can try to cope with the complexity by using models in the development process. In general a model is a dedicated abstraction of the real world. It can help to explain certain real-world phenomena [NL02]. Graphical representations of information play an important role in software engineering [GJ92, PH04]. They help to display the underlying data in a user-friendly way. Graphical modelling languages show the system on a very high level of abstraction. They can hide irrelevant data and clarify hierarchy in order to design the development process in such a way that it fits human understanding best. Good modelling suites join together capabilities of abstraction of human beings with the detailed implementation of machines. So they offer an intuitive model language to support human understanding and lead to an automated system code synthesis to support the target machines directly.

1.3.1 Different model types

The development of X-by-wire systems inherently need to study the communication techniques. Thus the communication subsystem should be part of the model if you develop a distributed real-time system. By integrating the architectural particulars of the distributed system into the model, Nossal and Lang [NL02] introduced a development process that categorizes four different model types: *functional model*, *architecture-allocated functional model*, *virtual prototype* and *architectural model*. These are classified according to the respective level of abstraction.

1.3.1.1 Functional model - discrete vs continuous

The functional model is the basic model that abstracts the system functions that are delivered to the environment. It models the main functionality of the system.

The functional model does not deal with architectural issues. Nor does it show whether the application will be distributed to several Electronic Control Units (ECUs) or run on a single ECU. Specification and allocation of tasks in the system are not part of it either.

In the functional model the continuous parts of the system are modeled in a data-flow language and the discrete part in state machines:

Continuous models Continuous models show the data flow between different operators of the model. The foundations of data flow models were laid by Kahn [Kah74] and a variety of such models are studied in [LP95].

Graphical models of such data flow consist of operators with specified input and output signals whose interconnections show the flow of data. If hierarchy is introduced, each operator itself is such a network of other operators which connect its inputs with its outputs. Basic mathematical functions are the smallest operators.

The idea of continuous data flow can only be approximately implemented on a computing machine, thus data flow models can have different types of *firing rules*, *i.e.*, the conditions that control the points in time where the data flow of each operator is evaluated. These conditions can be kept very flexible and even introduce nondeterminism in order to model the problems as accurately as possible. As nondeterminism introduces many problems for safety issues, *synchronous data flow* [LM87] evaluates the operators in a synchronous manner to predefined clock ticks.

Graphical notation and interface of data flow models are usually close to the control engineering culture and knowledge and enable engineers to model control loops in their systems.

Discrete models As a basic principle *transformational* and *reactive systems* build a dichotomy in the analysis of computing systems [HP85]. Transformational systems are described by a relation between input and output value. They are of linear structure with only the initial and the final state of interest. Reactive systems do not compute a function, but perform a continuous *interaction* with their environment. The behaviour of the environment, *i.e.*, the input to the system, depends on the reaction of the system, *i.e.*, of previous outputs. Therefore a relation between input sequences and output sequences is not enough to fully describe a reactive system. This phenomenon is known as the *Brock-Ackermann paradox* [BA81]. In this example two transformational systems perform the same operation on sequences. One behaves as a one-place and the other as a two-place buffer and they suddenly behave differently when their outputs are fed back to their inputs. This happens because the recycled output can appear earlier in the output sequence if a one-place buffer is used instead of the two-place buffer. Apparently, one also has to know *when* an output is produced [HdR91].

Harel introduced his *Statecharts* [Har87] to add the idea of being in a particular *state*. As mentioned before this internal state at which an input arrives is important for a reactive system to compute the reaction. The traditional finite state machine (FSM) does not serve this purpose because its only output is a

signal that it has reached its final state. In contrast a reactive system might not even have a final state and needs to be allowed to produce outputs at any time during execution. The solution is the formalism of the *Mealy machine* [HU79] which Harel extended for his needs.

Today a variety of different model languages based on this approach exist, which look quite similar but do not only have different names but different semantics. This goes back to the fact that desired properties of the statecharts, *i.e.*, *modularity*, *causality* and *responsiveness*, cannot come together [HG92]. Hence every modelling tool delivers its own statecharts derivate with semantics appropriate to the field of application.

1.3.1.2 Architecture-allocated functional model

The architecture-allocated functional model adds information to the functional model. This model goes beyond the state machines and control loops and takes the *system architecture* into account. It represents the allocation of parts of the modelled functions to architectural elements. The functions are grouped to tasks and then the tasks are allocated to ECUs.

The sole purpose of the architecture-allocated functional model is to concisely represent system distribution - that is hardware and software architectures.

Communication relations are modelled with native I/O operators of the tools, which operate without delay.

1.3.1.3 Virtual prototype

The virtual prototype is meant to accurately reflect the real-world system and therefore it must know which communication system will be used in the real-world system. It requires different models for TTP [KG92], CAN [ISO93], TTCAN [FMD+00], LIN [LIN00], byteflight [BPG00] and FlexRay [BMH+01], because different communication subsystems require different model blocks or nodes that represent, simulate and/or implement the different communication schemes. The virtual prototype comprises the functional behaviour of the system, the distribution and the communication system behaviour. The model includes

- the control loop model, as in the functional model;
- architectural information, as in the architecture-allocated functional model; and
- the communication system model.

After the development of the communication schedule based on the architectural models (described in 1.3.1.4) the architecture-allocated functional model can be augmented with the communication system behaviour. Therefore the native delay-free I/O operators are replaced by the communication-system-specific operators which are part of the communication system model. These new operators model specify communication system parameters.

The virtual prototype model assesses the influence of architectural decisions on a functional model. The functional model helps to find flaws in the control strategy, while the virtual prototype lets the developer adjust the algorithms according to specific properties of the communication subsystem. Reaction strategies for communication faults can be implemented and their effects simulated.

1.3.1.4 Architectural model

The architectural model focuses on a detailed description of the system architecture. The hardware part models the computational nodes (ECUs) and their interconnection networks (from a non-redundant bus to a multistar network topology). The software part describes the tasks with their relations, which are data passing and precedence. Developers augment the model with properties about the hardware and software, for example characteristics of the computers (*e.g.*, clock speed) and characteristics of task execution (*e.g.*, worst-case execution time).

The purpose of the architectural model is to configure the real-world system prior to runtime. It does not need to comprise the control loops and state machines because it does not use the functional behaviour. From the architectural model one can generate the static configuration data that is needed to run the real-world system: The model properties are exported to tools, which compute the static configuration data automatically or with little interaction with the developer. That is for example the communication schedule (Message Descriptor List (MeDL) in the TTP) and the component's operating system schedule if a TTA is used (Task Descriptor List (TaDL).

1.3.1.5 The A-process

The A-process [NL02] describes the development process from the beginning with the functional model to the final application code. Figure 1.10 depicts its workflow (and indicates the origin of its name). The developer begins by specifying the control loops in continuous and discrete models and hence builds the functional model. The architecture-allocated functional model derives from the functional model by the augmentation of architecture specific data, *i.e.*, the

information about the distribution of the system. The architectural model uses this distribution information and enriches it with detailed knowledge about the hardware and software properties from where middleware code (time triggered operating systems, fault-tolerant communication layer (FTCOM)) can be generated. On the other hand, the virtual prototype augments the architecture-allocated functional model by communication subsystem specific information and it uses the configuration data of the architectural model for simulation. Code generated from the virtual prototype together with the middleware code build the application code.

The A-process was introduced to help the needs of system developers: Specific purposes require specific information about the system. Therefore different models could be used. If different models were developed independently and represent different aspects of the same system, they can contradict each other in certain respects. A complete *virtual system model* could fix this issue: It would comprise all properties of the real-world system. Apparently this model would be very complex and incomprehensible. Thus the A-process introduces different models that are well coordinated and therefore consistent.

It must be evaluated, how the existing modelling tools support this approach. Some modelling suites offer a seamless development process that only allows modelling of a complete virtual system model instead of keeping the different model types apart.

1.3.2 Comparison of Modelling Suites

Several manufacturers offer powerful modelling suites that promise to provide the developer with tools for a seamless modelling process. In this Section the suites are introduced that are currently in use at CAU Kiel, 2005 (see Version information at Appendix C). Those are the *Matlab* family from *The Mathworks* with Matlab, Simulink & Stateflow and the *SCADE Suite* from *Esterel Technologies*. We will build a model of a time triggered architecture and both modelling suites will support this communication architecture by integrating the TTA tools of *TTTech* [TTT] (that use TTP as the communication subsystem) into their development workflow. The TTTech tools can be used on their own and they can use graphical representations for interacting with the developer. So they can be considered as modeling tools at a lower abstraction level. Therefore we will start with the introduction of the TTTech tools and then introduce the tools of Mathworks resp. Esterel Technologies.

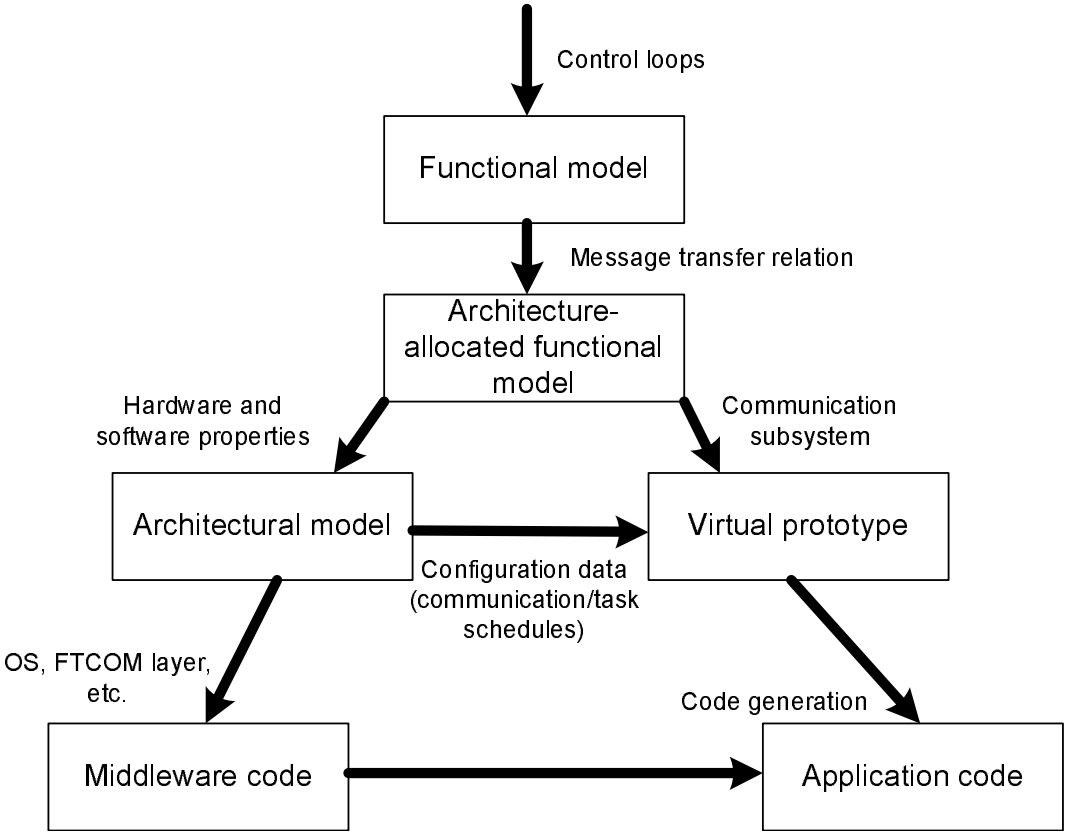


Figure 1.10: The A development process tracks the development of X-by-wire applications from functional model to middleware and application code

1.3.2.1 TTTech: Tools for specifying the TTA

The Time-Triggered Protocol, TTP, has been developed during the past 25 years by Vienna University of Technology (TU Wien) [KG92]. In 1998 *TTTech* emerged as a company from the TU Wien and EU-funded research projects such as TTA and X-By-Wire. From then on it has commercially been using the results by developing tools for the TTA system design. It has been trading dedicated TTP hardware, offering trainings on the TTP topic and their products and supporting development projects in the industry.

The target group are mainly the automotive industry and comparable industries, which shows a strong distribution of the development process into two parties: The *system integrator*, usually the car manufacturer itself, specifies the system and divides it into smaller subsystems. These are produced by *subsystem suppliers*, which often differ from the system integrator. In the end the system integrator integrates the several subsystems into the final system.

Therefore the TTTech tools build a two-level design framework taking this relationship between *subsystem supplier* and *system integrator* into account: TTP-Plan is an overall cluster design tool whereas TTP-Build is a node and task design tool.

Tool Overview

1. **TTP-Plan** Specifies global cluster properties. Generates global communication schedule—the Message Descriptor List (MeDL).

2. **TTP-Build** Specifies local node properties. Generates local task schedule—the Task Descriptor List (TaDL), configures the operating system, generates middle ware code, FTCOM Layer.

3. **TTP-Load** Loads application binaries and MeDL to the nodes with the help of the monitoring node.

4. **TTP-View** Connects the monitoring node to the cluster and listens to TTP traffic. Visualizes messages, records message traces.

5. **TTP-Calibrate** Uses monitoring node to calibrate application constants during runtime of the cluster.

TTP-Matlink Models a TTA in Matlab Simulink and exports the data to TTP-Plan and TTP-Build. Gives a user interface to control the other TTTech tools.

TTP-Plan TTP-Plan is used to specify the global TTP-cluster information. This information must be provided roughly in the following steps

1. Definition of global cluster information such as the *TDMA round period* and the *transmission speed* of the underlying physical layer.

2. Definition of the *hosts*, *i.e.*, the nodes used in the cluster, with rudimentary information as their serial number in the network for example.

3. Definition of the possible *cluster modes*.

4. Specification of the needed *subsystems* in the system that will be implemented by the hosts.

5. Mapping of the hosts to the subsystems according to the current cluster mode.

6. Definition of the *messages* that will be transferred by the communication subsystem including typing of the messages and defining the period.

7. Mapping of messages to the subsystems that send them.

The graphical user interface includes a step-by-step guide leading the user through the main steps of the specification process and a graphical interface, the so called *pilot*, that depicts the relationship between the elements and let the user directly access the corresponding specification forms (see Figure 1.11).

The tool comprises a consistency checker that looks for logical design faults.

After all the necessary information is provided by the system developer (optional data are filled with default values), TTP-Plan automatically generates a global static communication schedule, the Message Descriptor List (MeDL). It shows statistical data, such as the usage of the communication bandwidth (see Figure 1.12).

Then the developer can review the generated schedule and manually make changes to it like exchanging message precedence (see Figure 1.13).

The schedule information is stored in a so called *cluster database* which is the file format of TTTech for the cluster specification.

TTP-Build TTP-Build extends the development process of TTP-Plan by concentrating on the local host. While TTP-Plan is likely to be used by the system integrator who has to specify the necessary global information of the TTP-cluster, TTP-Build is interesting for the subsystem suppliers that have to implement one specific node of the cluster. Thus major steps of the design

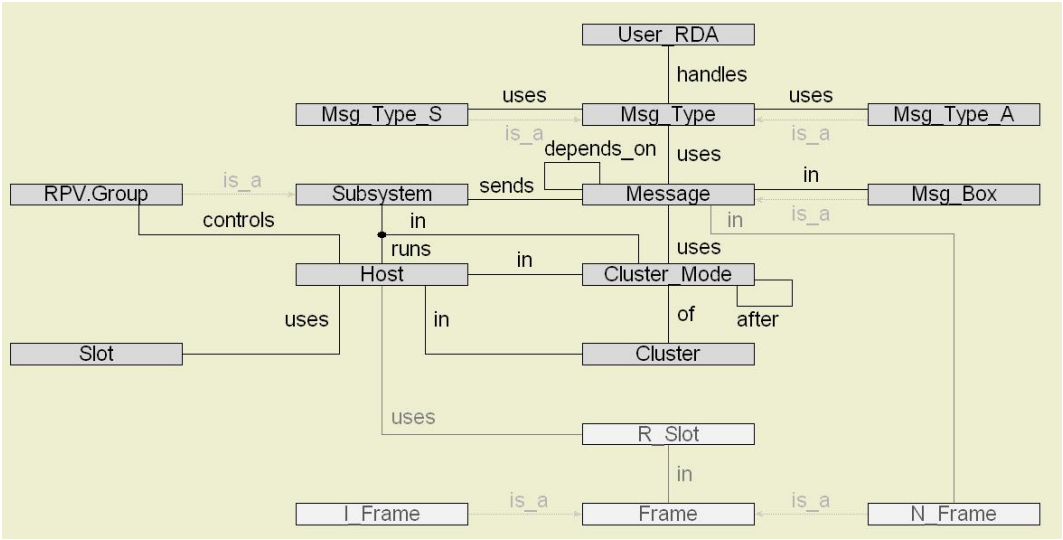


Figure 1.11: The *pilot* navigator of TTP-Plan that shows the relations in the global cluster specifications

```
Cluster_Mode__cluster___demoapp_powernode `demoapp_powernode_cm'
    TDMA rounds per cluster cycle          :        2
    TDMA round duration              (us)  :     1500
                stretch              (us)  :      690 (46.00%)
                inter-frame gaps     (us)  :      270 (18.00%)
                transmission time    (us)  :      540 (36.00%)
    kilobits/second                  (net) :      125
    messages/second                        :     1000
    message-instances/second               :     4000
    n-frames/second                        :     8000
    i-frames/second                        :    12000
```

Figure 1.12: Statistical data shown by TTP-Plan

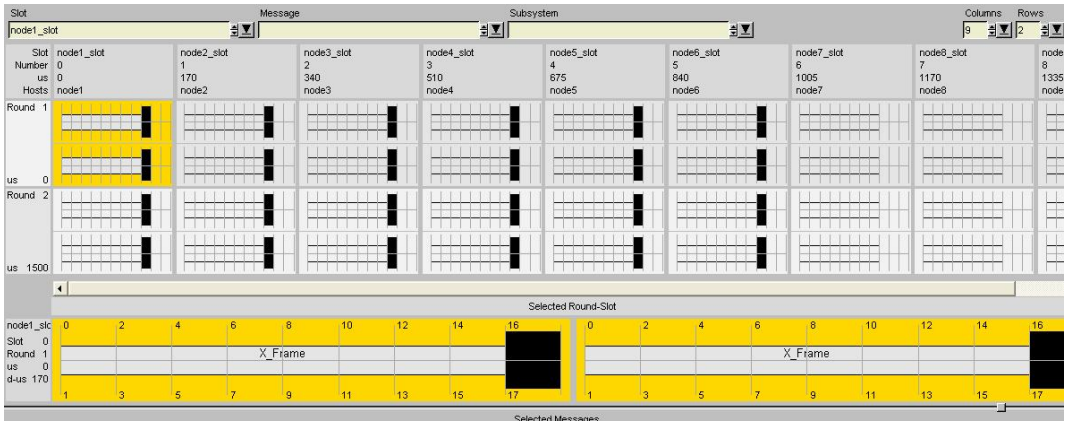


Figure 1.13: Graphical view of the generated global communication schedule in TTP-Plan

process are the configuration of the operating system and the generation of the fault-tolerant communication layer.

TTP-Build provides a step-by-step guide that leads through the main necessary specification steps and a *pilot* interface that graphically depicts the relations between the elements of the specification (see Figure 1.14), hence the graphical user interfaces of TTP-Plan and TTP-Build are very similar.

The specification process in TTP-Build roughly takes the following steps:

1. Loading of a cluster design. A cluster database that formerly was built by TTP-Plan has to be imported. The general node properties and the Message Descriptor List (MeDL) are inherited by TTP-Build.

2. The developer has to pick a specific node from a list of the available nodes in the cluster that he wants to specify. This shows that TTP-Build was mainly designed for a supplier that implements exactly one node in the cluster. If you want to specify more nodes of the cluster, you have to redo the TTP-Build steps again for every single node.

3. Specification of the node hardware. A node configuration must be loaded that specifies the TTP controller and the host computer hardware of the node. The selection is done from a list as the specific node configurations come from a configuration file that TTTech provides with the tools. Thus only supported hardware can be used with TTP-Build and for special hardware TTTech needs to adapt the configuration files.

4. Specification of tasks and the mapping of subsystems to tasks. Each task is run by exactly one subsystem but every subsystem may execute as many tasks as necessary.

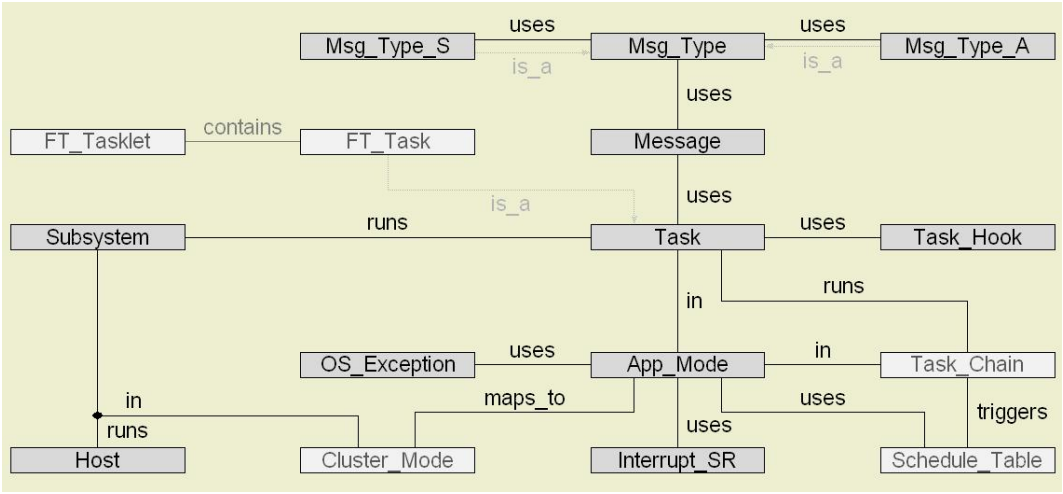


Figure 1.14: The *pilot* navigator of TTP-Build that shows the relations in the local node specifications

5. Configuration of the tasks: Setting of the granted time budget, *i.e.*, the *worst-case execution time (WCET)* of the task (see Section 3.1.3 for details on the topic of WCET). This is needed to build the task schedule for the operating system. Best-case execution times can be provided in order to let the scheduling algorithms optimize the checks and schedules.

6. Mapping of messages to tasks. Every message must be sent by exactly one task.

7. Definition of I/O messages. These are used whenever the node needs to communicate in a different way than via TTP. This especially includes access to the local I/O interfaces. For example regular local sensor readings of a node can be implemented by I/O messages. Property of such a message is its period. A task that is associated to an I/O message will be scheduled according to this period. Hence an I/O message can be regarded as a trigger for a task that is not scheduled according to any TTP message. Particulars of the I/O message must be manually implemented by the developer.

8. Mapping of tasks to application modes (which is only one by default).

After specification of the needed information, TTP-Build can generate the schedule for the time-triggered operating system of the node. This Task Descriptor List (TaDL) can be reviewed by the developer (see Figure 1.15).

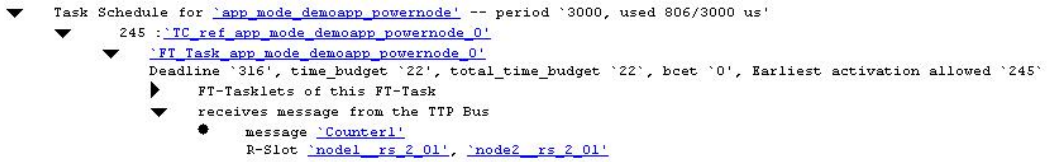


Figure 1.15: The TaDL generated by TTP-Build can be reviewed. It displays the timing information for each task and some details of the FTCOM-Layer

Then TTP-Build can automatically generate the fault-tolerant communication layer code and finally generate the full application code of the node. This includes the code for the operating system *TTP-OS*, the fault-tolerant communication layer code and templates for the several tasks of the code.

After this process the developer has to implement the task functionality in C within the templates. Afterwards he can compile the node sources and as a result the node binary is ready for deployment.

Other tools TTP-Plan and TTP-Build are the major instruments for the design of the system. They specify the system, build the schedules and generate the template application code.

There are some other tools of TTTech available that help with working with a TTA but they are not mandatory. So they are listed here for the sake of completeness. As *TTP-Matlink* and *TTP-Calibrate* are tools for Matlab Simulink, they will be introduced in Section 1.3.2.2.

Some of the applications use the TTP *monitoring node*. The monitoring node is a TTP node running special applications. It has an interface to the TTP network on the one hand and an Ethernet interface on the other hand. However, it is no gateway between TTP and Ethernet. The applications of the monitoring node only allow monitoring of the TTP network and loading of the node application binaries.

TTP-Load TTP-Load is a helper for the practical use of the generated application binary. It connects the developer's PC over Ethernet with the monitoring node and commands the monitoring node to connect to the TTP-cluster. The nodes in the cluster change their modes to the download mode and the so-called *bootloader* software module at the nodes takes over the communication control. The monitoring nodes checks the application binaries that are currently stored on the different nodes and checks if the MeDLs of the nodes fit together. Then the new MeDLs and application binaries can be uploaded to the nodes through the monitoring node. Besides the startup phase this is

the only communication mode where the access is managed asynchronously by the monitoring node.

TTP-View After uploading the new application code to the cluster, the TTP controller of the nodes switches back to the default TTP protocol and therefore automatically starts the time-triggered communication and executes the applications. The monitoring node can be used to see what is going on at the TTP-bus. It loads a cluster database and therefore knows what messages will be transferred. The current values of the messages can be displayed on the screen and visualized, for instance by gauges, bars or indicator leds. The status of messages can be displayed (for example how many messages of a redundantly transferred message which uses FTCOM are available) and the current membership vector.

The behaviour can be recorded over time. This record function can be triggered by specified conditions. This can be used if only faulty behaviour is meant to be recorded and the time till such a faulty behaviour is too long to record it all.

TTP-OS TTP-OS is the operating system that TTTech uses within their nodes. TTP-OS is based on the OSEKtime working group specifications for the OSEK/VDX operating system [OSE01]. TTP-OS is a bridge between the OSEK/VDX specification and the world of Time-Triggered Technology.

With generating the application template code for a node with TTP-Build, a fully configured version of TTP-OS is generated for this node.

1.3.2.2 The Mathworks: Matlab, Simulink & Friends

The Mathworks offers a big toolbox for the model designer. With the mathematical tool *Matlab* [Mat93] it provides a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numerical computation. Matlab forms the basis of all other tools. Main parts of the model-based system design are *Simulink* and *Stateflow*, whereas the *Realtime Workshop (RTW)* and the *RTW Embedded Coder* do the code generation. The plugin *TTP-Matlink* of TTTech is used for the TTA-coupling of Simulink. These parts of the Matlab family will be introduced in the following.

Simulink Simulink models data flow. It is used for simulation of dynamic systems. The creation of a model is done graphically by picking and positioning Simulink blocks from a block library and adding signal lines between the inputs

and outputs of the blocks. There is a big library of predefined blocks that implement mathematical functions. You may add your own blocks to the library.

The semantics of a data flow model are specified by its evaluation rules of the data flow operators. The simulation of a Simulink model uses *solvers* to set the evaluation behaviour.

Solvers are numerical integration algorithms that compute the system dynamics over time using information contained in the model. Simulink provides solvers to support the simulation of a broad range of systems, including continuous-time (analog), discrete-time (digital), hybrid (mixed-signal), and multirate systems of any size.

These solvers can simulate stiff systems and systems with state events, such as discontinuities, including instantaneous changes in system dynamics.

The developer can specify simulation options, including the type and properties of the solver, simulation start and stop times, and whether to load or save simulation data.

Explicit typing of variables is not mandatory in Simulink, but there exists a type-checking mechanism that might reject some models due to type errors. The boolean data type is added by a mapping to the numbers 0 and 1 and many quasi logical operators like switches accept any data type as input. The value gets internally compared to a threshold in order to determine the boolean value. Other operators usually handle multiple data types too, so there is the same operator for integers and floats. This can end up in a mixed use of types where even booleans can be typed as floats.

Stateflow Stateflow models state machines. Stateflow charts enable the graphical representation of hierarchical and parallel states and the event-driven transitions between them. Stateflow semantics are not formally given but by an informal description in its user manual [Mat04]. As explicit typing is missing in Simulink, some explicit mechanisms are missing in Stateflow as well: consider for example a state with multiple outgoing transitions. If some transition conditions are true in the same step, they are evaluated in a clockwise progression, starting at the upper left corner of the source state. This means that the position of states and transitions is critical to the semantics of the chart; if one slightly moves a transition, it could change the semantics. Therefore information such as source and target is not sufficient to describe the semantics of a transition.

In *Statemate* Statecharts or Safe State Machines of *Esterel Technologies* this is not the case: either multiple outgoing transitions are not allowed (Statecharts) or explicit ordering of the transitions is mandatory (Safe State Ma-

chines). Hence a chart will always have the same semantics, if the graphical representation is simply rearranged while the general relations between states and transitions are not changed.

Unfortunately Stateflow does not have this desirable property.

Code generation The Matlab family uses a set of tools to generate code out of the models. The *Real-Time Workshop (RTW)* is the basis for code generation of data flow models and the *RTW Embedded Coder* upgrades the RTW for building a code that is meant for embedded systems. Therefore the code is better readable, more compact and optimized for execution speed.

The result is ANSI/ISO C code and is generated incrementally for the model blocks. Therefore there are distinct files for all subsystems available.

Matlink Matlink is the interface between the TTTech tools and Simulink. As Simulink itself does not model distributed systems, Matlink brings this feature into the Matlab family.

Matlink provides a Simulink block library which is used to specify the TTA. There are blocks to specify the subsystems, tasks and messages of the TTA. Modelling of a TTA with Matlink is exemplarily shown in Figure 1.16. The example is provided along with Matlink by TTTech. It shows a brake-by-wire system that is specified by three subsystems for the pedal sensor, the brake force calculation and the brake force execution. Special blocks are used to model messages that are either sent or received by a subsystem. The next level is the specification of tasks as shown in figure 1.17 for the brake force calculation subsystem. This level specifies the tasks and local messages between the tasks. Additional forms allow to edit specific task properties, such as the time budget. The lowest level is shown in figure 1.18 where the brake force execution task is implemented. Standard Simulink blocks are used to model the tasks behaviour.

The special Matlink block seen in Figure 1.16 gives access to the other controls of Matlink. It opens a tabbed window which allows the developer to do final specifications of the TTA. This starts with global cluster settings such as the TDMA round length and the transmission speed of the physical layer and goes to the specification of the single hardware units, the nodes, and the mapping of subsystems to nodes. Finally it controls the other tools of TTTech:

- The developer can export the Simulink model into a cluster database and load it to TTP-Plan. All mandatory information about the cluster is derived from the Simulink model and therefore TTP-Plan can automatically build the MeDL for the cluster. An additional interactive modus allows to manually insert optional data to the cluster database in

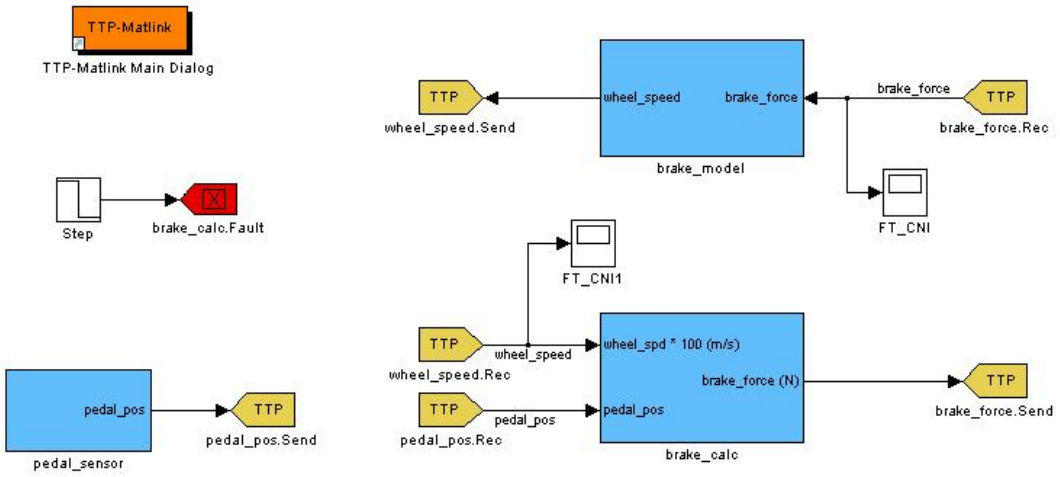


Figure 1.16: A brake-by-wire system modelled in Simulink with TTP-Matlink

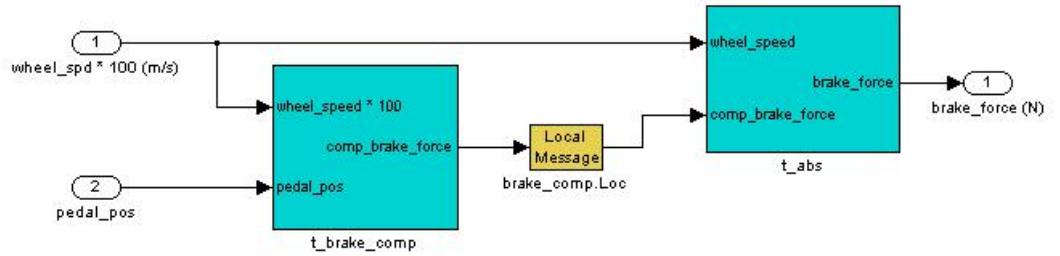


Figure 1.17: Specification of tasks within a subsystem (brake_calc) with TTP-Matlink

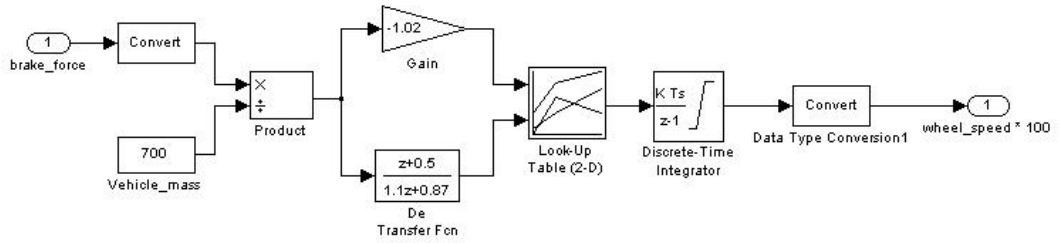


Figure 1.18: Implementing the functionality of a task (in subsystem brake_model) with standard Simulink blocks

TTP-Plan before generating the MeDL or manually edit the computed schedule.

- The schedule information can be imported into Matlink which allows the developer to see the timing information about the messages in the Simulink model.
- Then the enriched model can be exported to TTP-Build, that means the information for every node is exported to a single instance of TTP-Build. The task schedules (TaDL) for each node are made by TTP-Build.
- The timing information for the tasks can be imported into Matlink to show the task information in the Simulink model.
- The developer can make the toolchain generate the code for some or even all nodes. That means the generation of the operating system files, the fault-tolerant communication layer and the templates for the several tasks. If the developer modelled the tasks with Simulink blocks, the task templates are filled by the Realtime Workshop Embedded Coder with the appropriate implementation of the functionality of the task.
- The compilation of the generated code can be triggered for the nodes.
- The developer can call TTP-Load to automatically load and start the application binaries to the nodes.
- TTP-View can be called to view the cluster communication over the TTP-bus.

By centralizing these functions at the Matlink user interface, the modelling becomes a seamless process.

Calipo is a protocol developed by TTTech to calibrate constants in an application during runtime of a TTP cluster. *TTP-Calibrate* is a tool that includes this feature on the Matlink interface. Calipo (short for calibration protocol) adds a time slot in the MeDL for the monitoring node that enables it to send data. Constants in a model can be marked for calibration and then they are internally changed to variables whose values can be changed by the monitoring node. In this way the developer can define constants at the distributed nodes that set some special properties, brake force in a brake-by-wire application for example, and he can adjust these values in tests without the need of regeneration of the binaries and a new upload and restart of the TTP nodes.

With Matlink the dichotomy of system integrator domain and subsystem supplier domain vanishes. So it is likely be used for rapid prototyping where the developer has both roles.

1.3.2.3 Esterel Technologies: The SCADE Suite

The *SCADE Suite* is the modelling environment of *Esterel Technologies* [Est]. The semantics of the models are given by *synchronous languages*: A very common model in software process control is *cycle-based reaction*. The implementation cyclically repeats a sequence of three actions: reading the inputs, computing the reaction and producing the corresponding outputs. Input events occurring during a reaction are queued for the next reaction, which makes the reaction atomic and deterministic. There are two different synchronous programming styles: The data-flow style and the imperative style. These lead to the graphical modelling techniques of the SCADE Suite. An informal introduction to these synchronous languages gives [Ber00].

Data flow models *Lustre* [HCRP91] is the synchronous language with data-flow style which is well-adapted to steady process-control applications and to signal processing.

The following example is taken from [Ber00] and shall lead to the graphical representation in SCADE: Consider a dynamical system where U is the input signal, X the output and S an internal state variable with the equations

$$\begin{aligned} X_{t+1} &= U_{t+1} \cdot sin(X_t + S_{t+1} - S_t) \\ S_{t+1} &= cos(S_t + U_{t+1}). \end{aligned}$$

In Lustre the system becomes the following:

```
node Control (U : float) returns (X : float);
var S : float;
let
  X = 0. -> (U*sin(pre(X)+S-pre(S)));
  S = 1. -> cos(pre(S)+U);
tel
```

Note that the time indices are omitted and therefore the variables denote complete flows where the *pre* operator gives access to the value of the former instance and the `->` operator acts as initialization for the *pre* operator.

A SCADE model is a graphical representation of such a Lustre program. The example above looks like the graphical representation shown in Figure 1.19. The models are strongly-typed and available types are *bool*, *int* and *real* and the developer can combine these types in user-defined structures or enumerations. The operators in SCADE are called *nodes*. They can be modelled *graphically* as a net of connections between the inputs, outputs and other nodes as shown

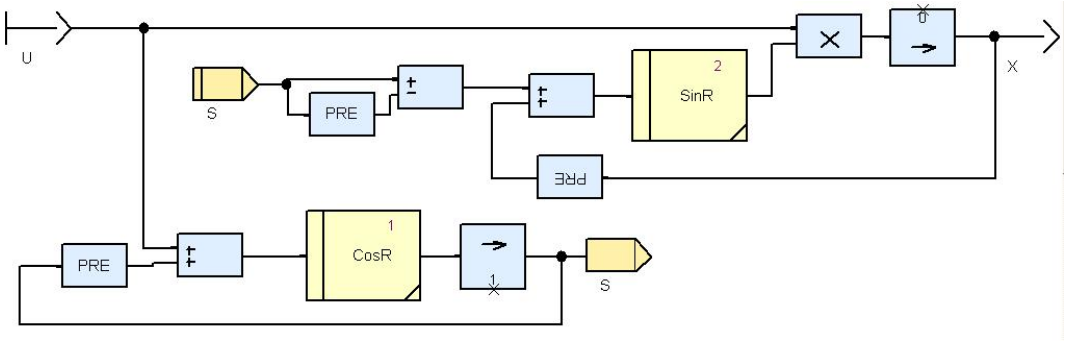


Figure 1.19: A simple SCADE model

in fig. 1.19 or *textually* with the Lustre language notation as shown above. Finally nodes can be indicated as *imported operators* which means they are implemented in the low level programming language C. SCADE provides a small library with standard nodes for mathematical functions.

Safe State Machines (SSM) State machines can be embedded into a SCADE model. Esterel Technologies uses its *Safe State Machines* which were formerly known as *SyncCharts*. They were conceived in the nineties [And96a] as a graphical notation for the synchronous *Esterel* language [BdS91]. Hence SSM inherit their semantics through the mapping to Esterel. This mathematical semantics are explained in a technical report [And96b] and an informal presentation of the model and its semantics is given in [And03, Han05].

Code generation SCADE provides Ada, standard C and qualified C code generators.

The SCADE Editor exports a textual description of the SCADE model which is fed into a *SCADE-to-Lustre* filter that transforms the model into Lustre code.

There are three different *Lustre-to-Code* transformators, that accept Lustre code as an input and produce code in the corresponding target language, *i.e.*, Ada, ANSI C or C qualified with respect to DO-178B level A [RTC92]. That is the most constraining level of one of the most constraining development processes, which is mandatory for safety critical systems in the aerospace industry. Esterel Technology abbreviates its qualified code generator with *KCG*.

Optimizations of speed, memory and code size can be enabled.

The generation of qualified code restricts the use of certain SCADE model components (for example the *followed by (FBY)* operator may not be used as it was introduced to SCADE after the verification of the KCG). The use

of a DO-178B template in the SCADE editor ensures that only models that are supported by the SCADE DO-178B qualified chain are used in the design phase.

TTP-Coupling The TTP-coupling to SCADE is called *TTPlink*. It offers the specification feature for the TTA to SCADE quite in the same way as Matlink does for Simulink: It provides a SCADE node library with nodes for TTP messages. There are nodes for modelling the sending of messages, the receiving of messages and the passing of local messages between tasks within one subsystem. Annotations to the message nodes add the message properties, such as the message names and the sending period. TTA subsystems and tasks are modelled by normal SCADE nodes which get enriched by annotations that include the subsystem resp. task properties. Forms allow the user-friendly input of these properties into the annotations. One standard SCADE node represents the TTA cluster. It only comprises TTA subsystem nodes and the message nodes that specify the communication between the subsystems. Each subsystem node only includes TTA task nodes that may be connected with local message nodes. The TTA task nodes may contain other SCADE nodes that model the functional behaviour of the task.

An additional tab in the file view is for specifying the hardware units, the TTP nodes, and the mapping of these nodes to subsystems.

Just as Matlink, the TTP-coupling for SCADE adds user interfaces to automatically generating the MeDL with TTP-Plan, the TaDLs for every node with TTP-Build, the wrapper code for the tasks and the code for the applications. The application binaries compilation and the upload of these to the TTP nodes can be called by the user interface as well. Finally the calibration of values through the monitoring node can be enabled by this user interface.

For examples see Section 2.3.2 where we describe the modelling of a real system and especially the modelling of the TTA in SCADE.

Additional features The SCADE Suite offers several additional features that should be mentioned.

Simulator A simulator can simulate the model by generating an executable simulation file for the model that will be executed by the simulator. Input signals can be set manually or by script files for larger traces.

Design Verifier Specified properties of SCADE nodes can be proved with the Design Verifier tool of the SCADE suite. A special verifier node library enables the developer to specify properties of his SCADE nodes within the standard SCADE notation. Model checking techniques test if the

model is in conformity with the properties and, if not, counter-examples are automatically generated. Using this feature is quite intuitive and requires no additional theoretical background.

Simulink and Stateflow gateways A Simulink gateway translates discrete controllers prototyped with Simulink into SCADE descriptions. A Stateflow gateway syntactically translates Stateflow charts into Safe State Machines (SSM). As the semantics of Stateflow and SSM differ, the translation results must be reviewed manually.

1.3.2.4 Conclusion

The synchronous languages that underlay SCADE and SSMs give precise mathematical semantics to the graphical models and therefore mathematical analysis of such models is possible and the behaviour is fully deterministic which is essential for safety-critical applications.

The deterministic, fault-tolerant philosophy that consistently inherits the time-triggered architecture perfectly fits the deterministic philosophy of synchronous languages. With its special features for validation, verification and qualified code generation and the underlaying use of the synchronous languages Lustre and Esterel the SCADE Suite goes consummately together with the time-triggered architecture.

Therefore we will use the SCADE Suite for our modelling activities introduced in the next Sections.

1.4 The DECOS Project

DECOS stands for *Dependable Embedded Components and Systems* and is a EU funded Integrated Project (IP) of the 6th Framework Programme (FP) [DEC]. It is domain independent. Possible applications are for example in the automotive, aerospace, railway, control and medical domain.

It tackles five key obstacles in the development of real-time systems (cp. [Kop03]):

Electronic hardware cost Today's distributed systems are *federated* to stay safe. In a *federated architecture* functional and spatial separation goes together. For example in a modern airplane every function gets its own hardware unit in order to keep any faults to this unit only. With the growth of functionality, hardware costs increase and space, weight and complexity requirements as well.

Diagnosis and maintenance If a fault occurs, it is hard to find the correct reasons. Diagnosis of today's electronic systems becomes a guessing game

for the maintenance staff and in the end complete Electronic Control Units (ECU) are exchanged because the fault could not be localized exactly. As the fault origin is often not found, the fault is not fed back into the design cycle in order to improve the system.

Dependability With the advance of electronic systems into safety critical applications, their dependability must be ensured. This includes safety, reliability, maintainability, availability and security.

Development costs Implementing more features while decreasing the development costs are often critical to stay competitive in industry. Thus reusability and fast development processes by the introduction of tool usage must reduce the development costs of single systems.

Intellectual property design A development process must consider the interests of subsystem suppliers who do not want to openly provide their subsystem code to the system integrator freely open. Usually they provide only binaries that execute the required functions but cannot be reviewed. The DECOS project will secure the intellectual property of the supplier.

Rushby summarizes the topic of federated and integrated systems as follows:

> *Systems used in safety-critical applications have traditionally been federated, meaning that each "function" (e.g., autopilot or autothrottle in an aircraft, and brakes or suspension in a car) has its own fault-tolerant embedded control system with only minor interconnections to the systems of other functions. This provides a strong barrier to fault propagation: because the systems supporting different functions do not share resources, the failure of one function has little effect on the continued operation of others. The federated approach is expensive, however (because each function has its own replicated system), so recent applications are moving toward more integrated solutions in which some resources are shared across different functions. The new danger here is that faults may propagate from one function to another; partitioning is the problem of restoring to integrated systems the strong defences against fault propagation that are naturally present in federated systems.* [Rus01]

This is the way to go: Change from a federated distributed architecture as described above to an integrated one. Why use one hardware unit for each function if the computing power of the units allows us to execute more than

one function on the unit. The new approach is meant to provide an integrated distributed execution platform.

It shall comprise pre-validated hardware and software components, suitable for generic use, because the reinvention and validation for every new function needs a lot of effort and money.

A seamless tool chain shall enhance the development process of dependable embedded systems. Tools can help to shorten the development process by putting the process onto a higher level of abstraction while they automatically take care of the lower level implementations. Model checkers and testing tools can help to find errors more quickly and modelling tools model system specifications to find errors in early design phases.

Reusable software modules used as operating systems and middle ware code help to decrease redundant development.

The tight integration of software by strict memory and processor usage separation leads to hardware units that concurrently run applications from different vendors and with different criticality levels [Kop04].

Such hardware and software units have already been developed and were deployed in the new Airbus A380 airplane as so called *integrated modular avionics* (*IMA*).

The time-triggered communication approach has not been widely used in industry yet as it is a relatively new technology and only little cheap hardware for the mass market is available. The development effort is quite high if the application exceeds the features of the current existing tools of *TTTech* (see Section 1.3.2.1 at page 20). So industry still uses cheap event-triggered solutions such as *CAN*, where the development effort seems to be little, or very expensive proprietary but safe time-triggered solutions like *ARINC 659*, also known as the *SafeBus* [Rus01], for those aerospace applications where guaranteed safety is absolutely needed.

DECOS aims to develop a generic execution platform and a toolchain in order to provide a distributed architecture that enables the developer to build highly dependable systems with relatively little development effort in order to push the time-triggered approach with its safety enabling features into industry.

1.4.1 The Aerospace Subproject

To validate the achieved results and demonstrate their practicability in a highly safety-critical application environment a test bench will be developed (a high-lift flap demonstrator).

The demonstrator will implement an electronically synchronized *high-lift flap system*. This system is motivated by the current state-of-the-art for such

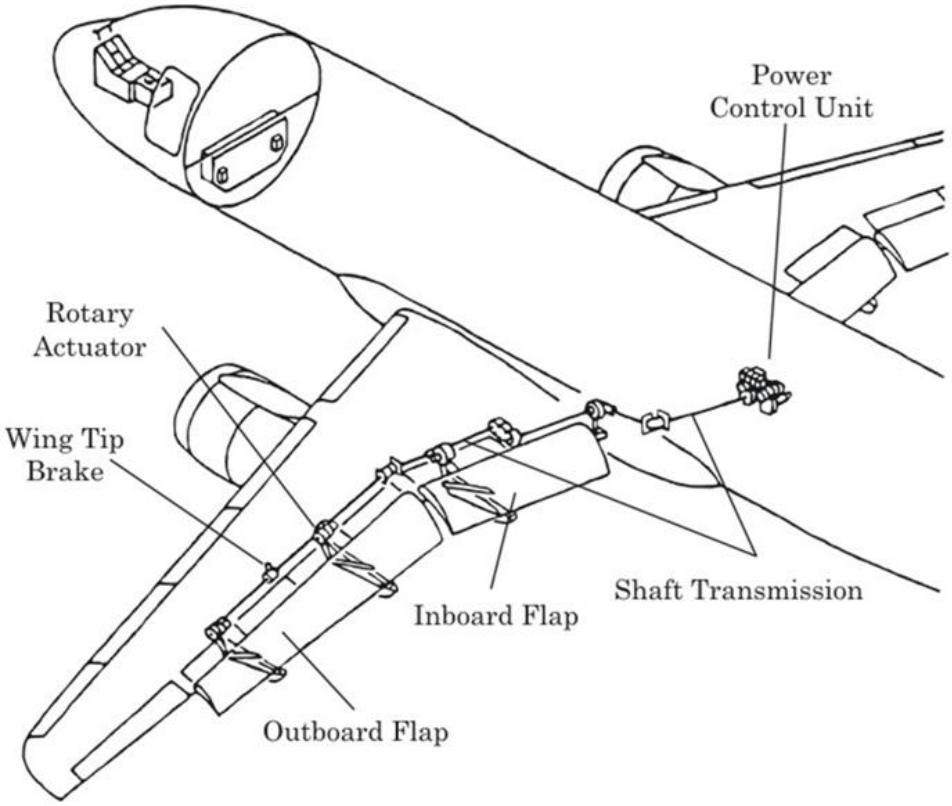


Figure 1.20: The state-of-the-art high-lift flap system. A mechanical tip-to-tip shaft transmission across the fuselage guarantees synchronization. Source: TUHH

systems (cp. [Neu02]): As depicted in Figure 1.20, a flap system consists of two flap panels at each wing. All flaps are connected by a tip-to-tip shaft transmission across the fuselage. A central power control unit in the wheel well actuates the flaps. For redundancy reasons the power control unit comprises two electric motors that move the shaft by a speed summing differential. In case of a critical fault a hydraulic brake—the so-called wing tip brake—can lock down the flap shafts and therefore the system is disabled.

The lift of an airplane increases as the square of the flying speed. Therefore a plane must take off and land at a very high speed. To avoid this, the flap system can increase the concavity of the wing if activated and therefore increase the ascending force temporarily. This is used at low speed for landing and take off purposes only. Hence a flap system is safety-critical. If the system fails during the flight, it will not be available for landing, which is obviously a serious problem. It is an even worse scenario if the left and right flap panel are not perfectly synchronized and hence the ascending force on the two wings

differ. This compromises the controllability of the plane and might ultimately lead to a crash.

To avoid the asynchronous state of the flap panels, they need to be synchronized. In the state-of-the-art flap system this is done mechanically: A tight mechanical shaft physically connects the left wing flap panel with the right wing flap panel. This shaft lies across the whole aircraft fuselage. The central power control unit actuates the shaft with the help of two electrical motors. If one motor fails, the other side will move the whole shaft with half speed. This way both sides are always perfectly synchronized unless a shaft breaks or blocks, whereupon the shaft brakes freeze the system. However, this scenario is highly unlikely.

The drawback of this solution is obvious: The shaft across the fuselage is very inflexible and the development of the tip-to-tip shaft transmission is very laborious and includes the internal construction of the fuselage into the development of the flap system, which makes the system neither modular nor reusable.

Weight is critical in the aerospace domain; any weight saved allows to carry either more fuel or more passengers, which means increased revenues for the airlines.

An electronically synchronized flap system could give more flexibility by introducing modularity and might even increase safety by exchanging central control with local controls enabling more sophisticated fault reaction strategies. Additionally it could save weight and space, but this is dependent on wiring. An electronic system has not yet been tackled, as safety has highest priority.

The demonstrator is a test platform for the aerospace domain to realize the implementation, test and integration of tools, methods and components (hardware and software) being developed in the DECOS project and to validate the achieved results and demonstrate their practicability in this highly safety-critical application environment.

Figure 1.21 shows the architecture of the system. Only one flap panel per wing side is shown. The left and right side of each flap panel are still connected via a mechanical shaft. Each shaft side is powered by a powerful electronic motor. The motor is controlled via the *motor control electronics* (*MCE*) whereas *actuator control electronics* (*ACE*) control the synchronization process and form an outer control loop. The ACEs are implemented in two parts: a control part and a monitor part for safety reasons. *Position pickoff units* (*PPU*) measure the current angle of the flap shaft and a hydraulic *cross shaft brake* (*CSB*) is able to fix the shaft in case of faults. The ACEs and PPUs communicate via a time-triggered bus in order to exchange information for the synchronization control loop. Both the control and the monitoring part of the ACE have their own interface to the TTA bus for redundancy reasons. The

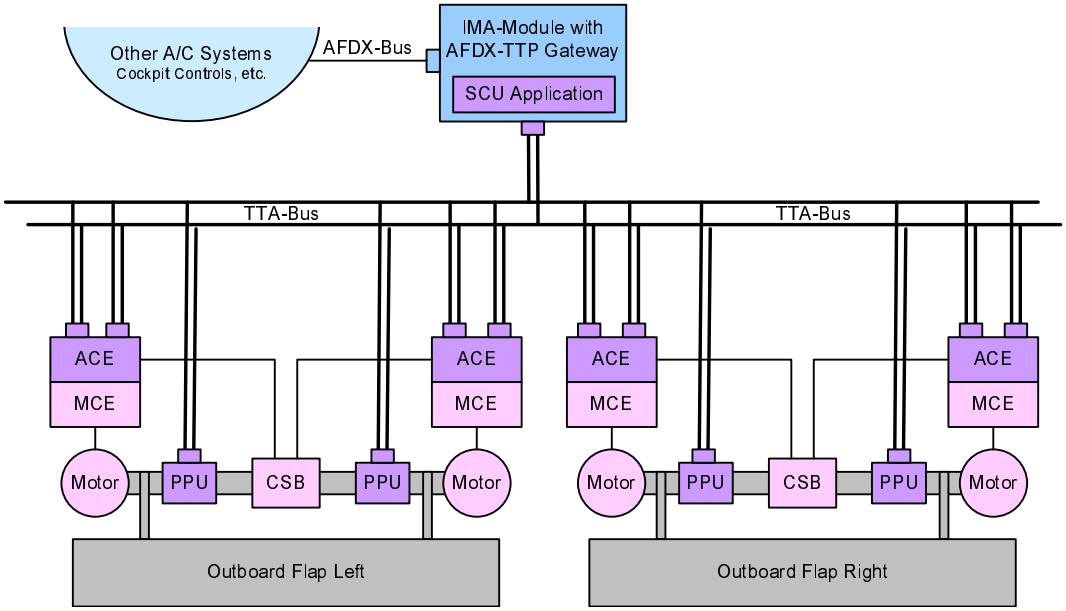


Figure 1.21: The architecture of a possible real electronically synchronized flap system

ACEs control the CSB and only if both ACEs on each side indicate correct functionality, the CSB is released.

A central *system control unit* (*SCU*) monitors the behaviour of the system and commands the desired flap angles. The SCU is implemented as an application on an IMA module which has the interface to the TTA bus on the one hand and the interface to the event triggered communication bus *AFDX* (Avionic Full Duplex Switched Ethernet) [ARI] on the other hand. Through the AFDX channel other aircraft systems are able to communicate with the SCU application and could even communicate directly with the TTA nodes by the AFDX-TTP-gateway functionality.

1.4.1.1 The test bench

A physical test rig of a flap panel exists at the Technical University Hamburg-Harburg (TUHH). The available part is depicted in Figures 1.22 and 1.23.

The rest of the system will be developed around this test rig. Hence the real system architecture will look as shown in Figure 1.24. The test rig will implement the right flap side. The mechanical parts will be augmented by the electronic units as described above. The left flap side will be implemented as a real-time simulation: The ACEs will be available as electronic units but con-

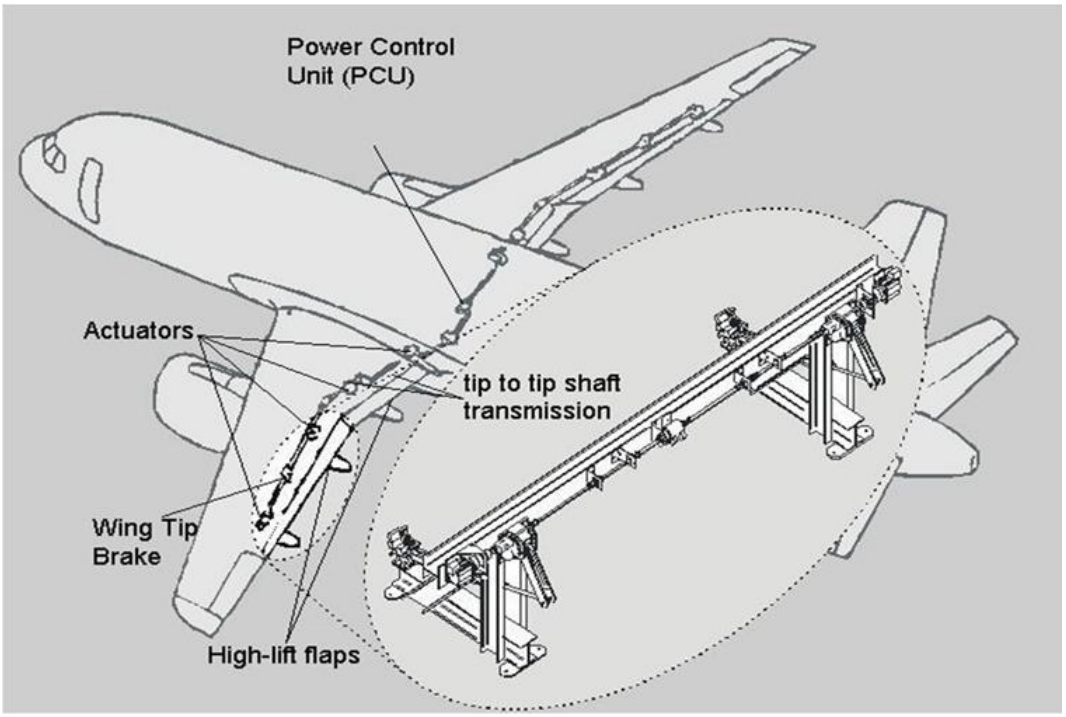


Figure 1.22: The available test rig is the mechanical part of one flap panel. Source: TUHH

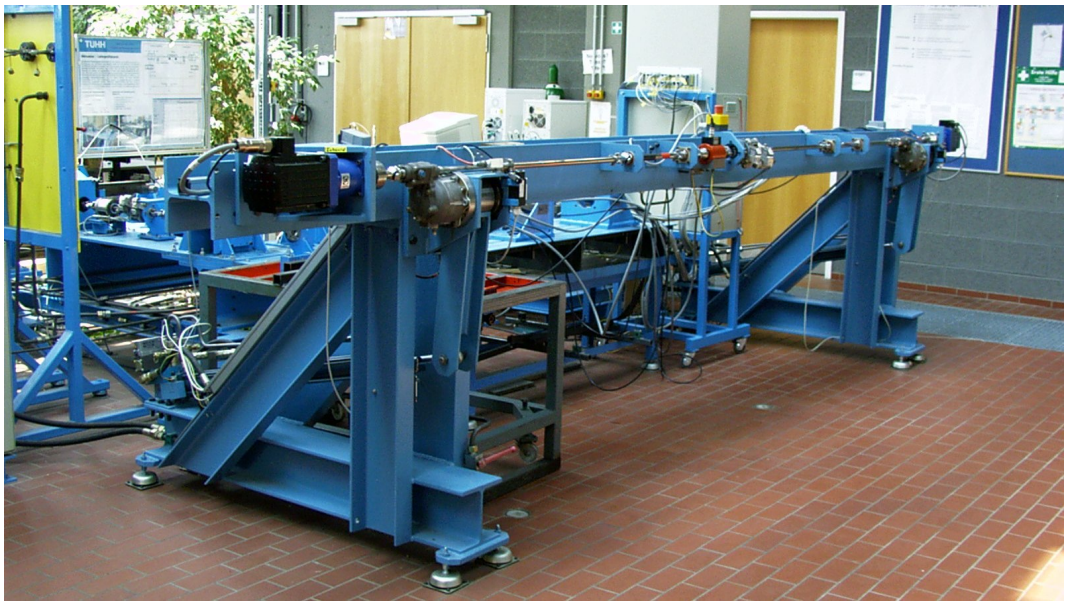


Figure 1.23: The physical test rig of TUHH. Photo: TUHH

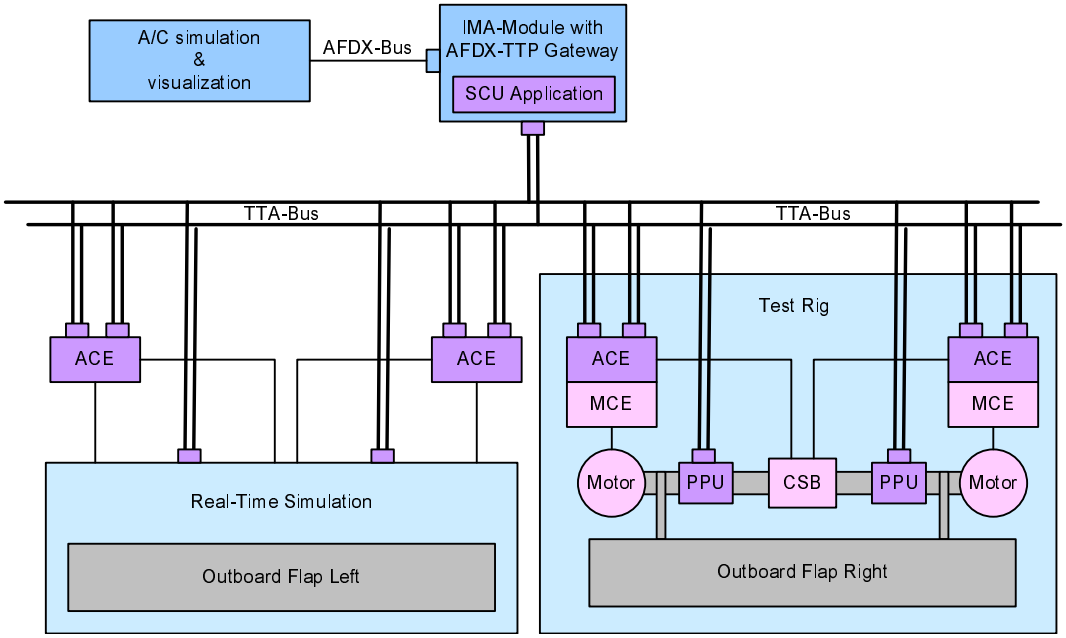


Figure 1.24: The architecture of the flap system developed around the flap test rig

nect to a computer unit that simulates the behaviour of the left flap. The SCU will be implemented on a IMA module as described but the input commands come from an aircraft simulation and visualization application at a personal computer that connects it via AFDX interface to the IMA module.

1.5 The RISE Project

RISE is another European IST project [RIS]. RISE stands for *Reliable Innovative Software for Embedded Systems*. It researches concepts and methods for the development of embedded reactive software, in particular for cars.

The project sets up a toolset addressing the phases: *Design*, *formal verification* and *automatic code generation*. The toolset will rely on the TTA for distribution of the code, on the SCADE Suite for design, formal verification of safety properties and automatic code generation and on Simulink/Stateflow for the design of control laws and algorithms.

The results of the project include

- A Simulink-to-SCADE Gateway,
- A Stateflow-to-SSM Gateway and

- A coupling to the TTA.

We are not directly connected with the RISE project. However, some project partners, such as *Esterel Technologies* and *TTTech*, participate in both the DECOS and the RISE project. Therefore the results of the RISE project are provided to us. Therefore, we are able to use the SCADE Suite with all the features described in Section 1.3.2.3. The TTA coupling and the gateways are of great impact as they exactly fit our needs.

1.6 Related Work

Main ideas of the Time-Triggered Protocol (TTP) evolved out of the MARS (Maintainable Real-Time Systems) project [KLMP82] at the Technical University of Vienna which started in the early 80s. Since Hermann Kopetz introduced the Time-Triggered Protocol (TTP) in 1992 [KG92] a lot has happened around it. Many of the properties have been analysed: Merceron et al. [MMP98] used synchronous programming to specify TTP while Nossal and Lang [NL94] used the formalism of Petri Nets. The clock synchronisation algorithm has been studied by Pfeifer et al. [PSvH99] while Merceron et al. [MMP99] showed that TTP is free of collision in normal mode. Merceron [Mer01] also tackled the membership algorithm and proved that TTP avoids multiple cliques. TTP was compared with other communication architectures by Rushby [Rus01]. Hexel [Hex03] developed a fault injection architecture and extensively analysed TTP. A number of small design problems were found and subsequently corrected in the protocol.

With the *cluster compiler* [KN95] a powerful application appeared that simplified the development of *time-triggered architectures* [KB03]. It ultimately found application in the commercial tools of TTTech [TTT].

Since "model-based" system development is a broad notion, there are a lot of developments around that topic. See for example the activities of Lee's [Lee03] and Sangiovanni-Vincentelli's [BWH+03] groups at UC Berkeley [Uni] or Jähnichen's group at TU Berlin [Ste].

So far it seems as if *TTA* and *model-based system design* only merge together in the tools, but the result does not elaborately get analysed. [NL02] describes details of a model-based development process for X-by-wire applications. Although it includes the communication scheme, the focus is the development process in principle. The modelling suites that start to support the TTA look more like a front-end to the cluster compiler in *TTP-Plan* (see Section 1.3.2.1). They hardly add new features but specify the TTA in a more convenient way than TTP-Plan itself.

As mentioned before, this work is carried out in the context of the DECOS project. DECOS builds on the foundations laid in previous European research projects (NextTTA, FIT, TTA, SETTA, RISE, X-By-Wire, PDCS, DEVA, DSOS), which took the TTA to the technical state of today. Browse the DECOS web site [DEC] to find detailed information about the related projects.

2 Modelling the High-Lift Flap Test Bench

2.1 Purpose of the model

There are many reasons that justify to use modelling and to make a system model to a central part in the development process of the system:

System specification Specifications are often written in natural language and are therefore of informal character. Models typically have a precise semantics (depending on the modelling language used), hence transferring the textual descriptions into a model formalizes a specification. As models usually show good capabilities of abstracting hierarchy, one would probably understand the essence of a system more quickly by studying a model than by reading a textual specification.

Computing the communication schedule Before the different components of the TTA can finally be implemented, the communication schedule - Message Descriptor List (MeDL) - must be defined, because the component schedules are built according to the MeDL. Modelling tools, such as Matlab/Simulink and SCADE, offer interfaces to the TTA cluster compiling tools of TTTech. So the MeDL can be synthesized directly from the model. Modeling the TTA in an appropriate tool is much more intuitional and better to read as working with TTP-Plan directly.

Specifying logical behaviour As an electronically synchronized high-lift flap system is a new concept, there are no synchronization algorithms and fault reaction strategies available from the shelf. So the development of the logical functionality of the different components must be done from scratch. Building a model and simulating its behaviour might help to find possible failure modes and assist at the development of this logical behaviour.

Test, verification Many modelling languages bestow models with such precise semantics that the model can be simulated. Hence the developers are enabled to test the system before any physical implementation is

available. Therefore elimination of design faults might be done in early stages of the development process, which saves development costs. Additional verification techniques boost this effect: with SCADE and the Design Verifier the developer can prove predefined properties of the operators and *Model Coverage Analysis* verifies that every element of the model (which represents a software requirement) has been dynamically activated when the system requirements are exercised. A primary objective is to detect unintended functions in the software requirements. Model coverage analysis is not yet supported by the SCADE current version (see Appendix C), but might be used in later stages if it becomes available.

Code generation Powerful modelling suites like SCADE and Matlab are equipped with code generators that transform the models into C and in the case of SCADE even in C or Ada source code. This code can be compiled with specific target platform compilers and loaded to the system components directly or with little hand coded additions. Having a model in such detail that it allows the synthesis of the program code of the components would be a great benefit of the modelling effort.

2.2 Scope of the model

The main scope of the modelling is the time-triggered communication. So all components that have access to the TTA bus are modelled at first. These are

- The four *Actuator Control Units* (*ACE*) which control the motor control electronics and therefore directly influence the motor motion which moves the flap panel.
- The four *Position Pickoff Units* (*PPU*) that detect the current position of the flap panel and provide the values on the TTA bus.
- The *System Control Unit* (*SCU*) that gets positioning commands from the cockpit resp. the aircraft simulation via the AFDX bus.

For simulation traces that provide significant results, the model has to be enriched by more detailed information. As the main purpose of the system is to synchronize the left and right flap panel, it is apparently reasonable to model the flap panels, too. Although the mechanical functionality is not the main scope of this model, it would be very helpful to have such information about the panels. This includes the functionality of the *Motor Control Electronics* (*MCE*) and the mechanical, electric and hydraulic parts of the flaps, *i.e.*, the

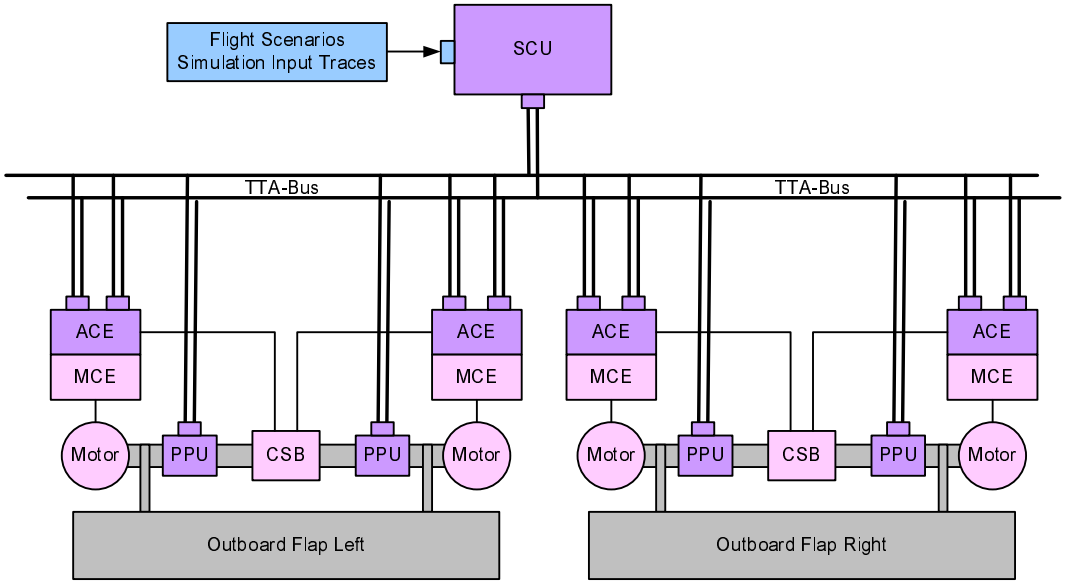


Figure 2.1: Scope of the model

main shaft, beds, the *cross shaft brake* (*CSB*), the electric motors and the mechanical connection of the PPUs.

Our intended purposes of the model do not require to model the AFDX-TTP-Gateway and the details of the IMA module that houses the SCU software. Hence they are not in scope of this model. The SCU software will only be modelled by its logical behaviour. The cockpit commands coming via the AFDX bus to the SCU will be implemented by textual flight scenario traces that can be fed into the simulation directly to the SCU. A trace format for these files must be worked out with the project partners in order to use the same flight scenario traces for the simulation of the model as for the simulation of the real test bench.

Figure 2.1 shows the scope of the model.

2.3 A simplified model in SCADE

When we started to work with the SCADE Suite, the TTP coupling has not yet been available. Following the *A-process* depicted in Section 1.3.1.5, this was no problem. This process starts with the *functional model* that implements the control loops and neglects the distribution of the system.

Next step in the A-process is the *architecture-allocated functional model* that is meant to concisely represent the system distribution. It groups functions together to tasks and allocates tasks to ECUs.

The TTP communication scheme finds application in the *virtual prototype* and the *architectural model*.

We assessed the SCADE Suite and the system requirements in order to discover whether the A-process is applicable in this dimension or not.

The knowledge about the system rested on the requirements specification [Koc04]. This does not provide much information about the control loops and any logical behaviour of the components. It only comprises little information about the functionality of the system and includes some information about the distribution of the system. Additionally it only provides rudimentary information about the TTP communication. Some messages were specified, but the project partners explicitly said that the final message information will be devised in the specification phase of the project. Hence new messages could be introduced or known messages could be omitted in the future.

The result was that we combined the four models and get only two:

Functional model The first model is the *architecture-allocated functional model*. The logical behaviour of the model will be developed by the project partners in the specification phase. Therefore it was not possible to implement much of it. As the distribution information was available, we decided to start modelling this information and feed the model with functional information by and by. We call this model the *functional model* for convenience.

Virtual prototype The second model is the *virtual prototype*. When the TTP coupling for SCADE arrived, we started to add the TTP items to the first model. There is no special tool that could be used to provide the architectural information that is designated for the *architectural model*. There is no need for a special tool since we see the cluster database of TTP-Plan as our architectural model, because it comprises all the architectural information that is needed to produce the Message Descriptor List (MeDL) for the system. Most of the architectural information will be fed into the TTPlink interface of SCADE and then the cluster database gets automatically exported to TTP-Plan as described in Section 1.3.2.3. Therefore the *architectural model* fuses into the *virtual prototype*.

2.3.1 The functional model

We wanted to be able to simulate our functional model with the SCADE simulator. As no detailed information about the logical behaviour of the components was available, we developed a simplified model using basic linear functions. Variables that have to do with advanced functionality were fixed to

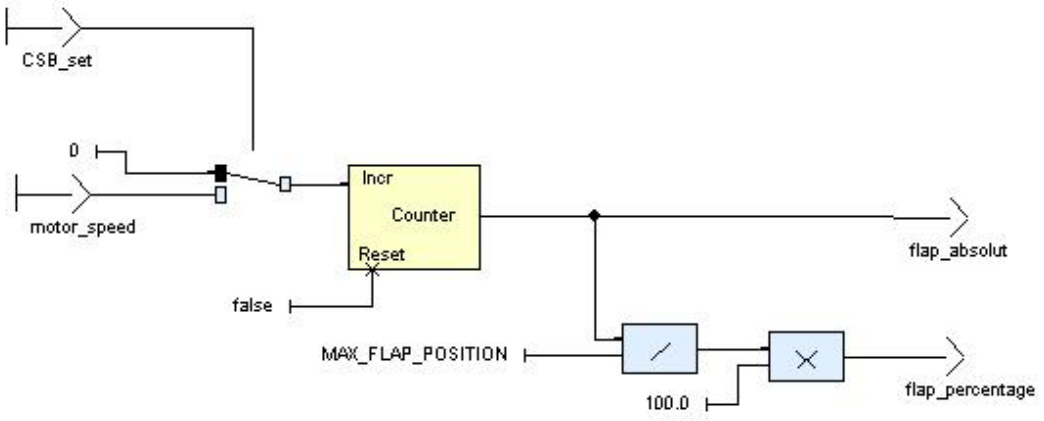


Figure 2.2: The rudimentary abstraction of the flap panel

default values and only the rudimentary functionality of the system was implemented.

The flap panel The physical flap panel is the system in the environment that our electronic system must control. In order to get simulated, we needed a model of the flap panel itself. In the simplest form it is just a counter that represents the current position of the flap. The first model of the flap is shown in Figure 2.2. The counter is changed by the `motor_speed` input that denotes the revolutions per minute (RPM) of the motor. The resulting value is an integer that shows the current absolute position of the flap. The `MAX_FLAP_POSITION` denotes the value at its possible maximum. That comes from the required resolution of one shaft rotation and the requirement to distinguish up to 400 rotations. So next to the absolute value, the flap outputs a percentage for better readability of the current position.

If the cross shaft brake is active, no motor can move the flap. Therefore the increment of the counter is set to zero.

This model is likely be replaced in the future by a more accurate model of all mechanical parts. This model is developed by the Technical University Hamburg-Harburg (TUHH) in Simulink and will be incorporated into our system model.

Inner control loop The *position pickoff unit (PPU)* and the *actuator control electronic (ACE)* form a control loop. Figure 2.3 shows the model and especially the data flow. This will be explained from left to right:

- With multiplexers and demultiplexers the signals are grouped in order

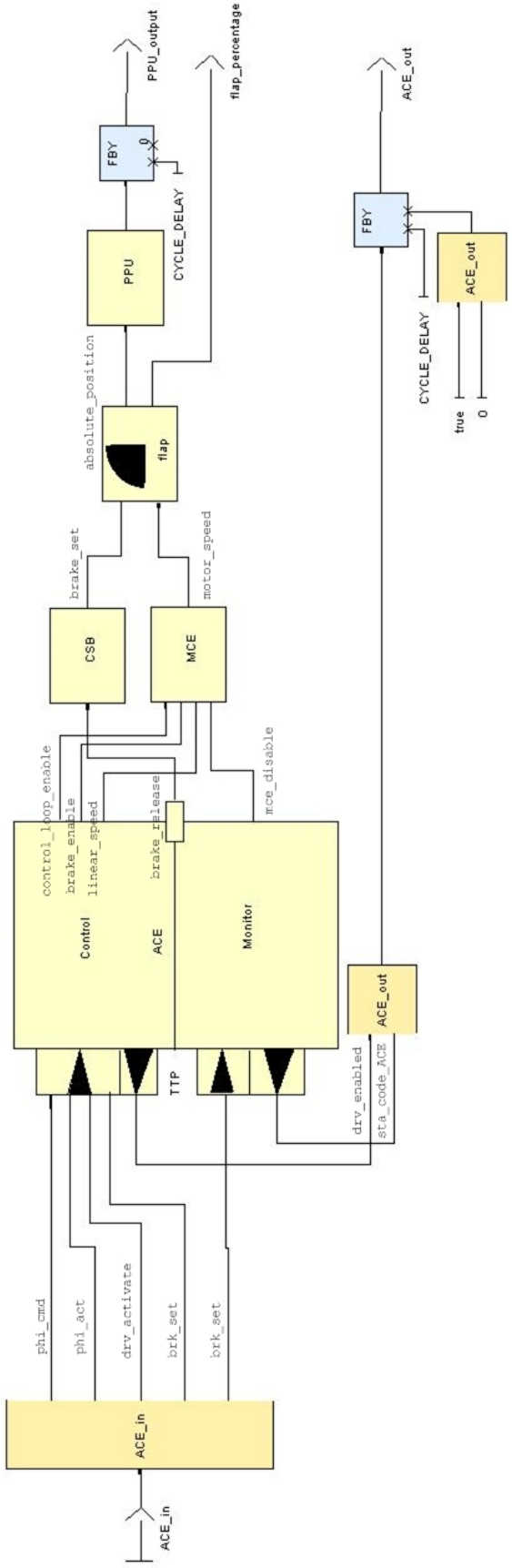


Figure 2.3: The parts of the inner control loop of the system

to form a clear interface.

- The ACE has two independent parts: A *control* part and a *monitor* part. According to the requirements both have their own TTP controller.

- The control part reads the `phi_cmd` that inputs the target value of the flap from the *system control unit (SCU)*. It gets the actual value of the flap from the PPU and additional commands from the SCU that activate the ACE and command to set the CSB. The monitor part reads the brake command of the SCU and maybe some other signals that are not yet specified, since the functionality of the monitor channel is not fully explained by the requirements document.

- The ACE outputs a status code and a Boolean value to the TTP network. The boolean value indicates whether the ACE is enabled.

- The ACE commands the *motor control electronics (MCE)*. The control channel can enable the control loop and command the MCE to brake by blocking the motor. A linear speed command to the MCE sets the direction and speed of the motors. The monitor channel can suspend the MCE in case it detects some failure.

- Both control and monitor part of the ACE must agree to release the CSB.

- The MCE model includes the motor itself, so it outputs the motor speed as its RPM.

- The cross shaft brake is simply represented by a boolean value that is either set or not. This value is fed into the flap model.

- The flap was described above and therefore takes the CSB status and the motor speed and outputs the absolute and the relative position of the flap.

- The PPU reads the absolute flap position and feeds it back to the TTP network.

- Some of the outputs of this model are fed back into the control loop as inputs of the ACE. Therefore we must introduce a delay with the `pre` or `fby` (followed-by) operator in order to avoid instantaneous feed back.

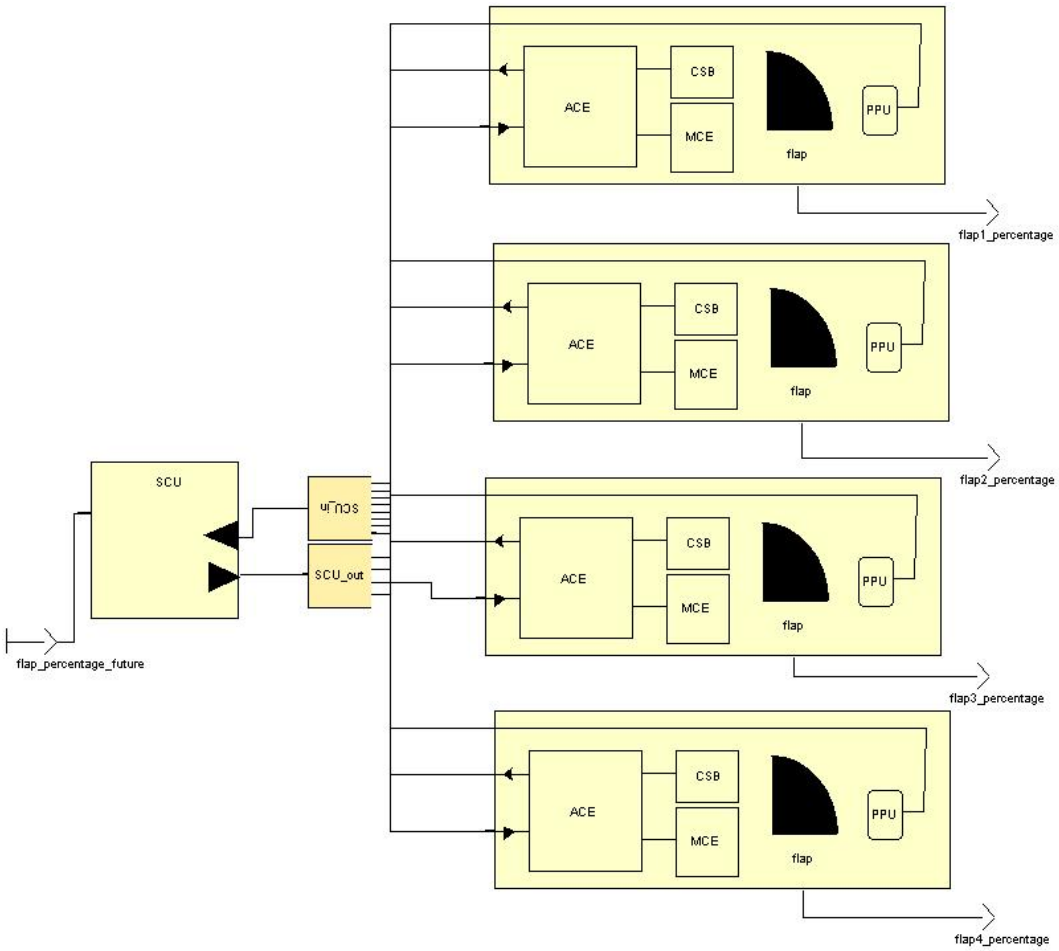


Figure 2.4: The inner control loops get connected with the SCU

Outer control loop A bus cannot be modelled with the standard data flow techniques in SCADE. From the requirements document it is not clear which components read which messages of the TTP bus. To obtain a complete control loop including the SCU, we therefore connected the four inner loops with the SCU directly. We used the standard delay-free connection of SCADE. Hence we abstracted from communication delay unless the former introduced delay operators to prevent recursion.

The model is shown in Figure 2.4. Although we used structures for grouping the communication signals, the data flow gets quite complex. Introducing TTP message nodes in the virtual prototype will alleviate this.

We can simulate this primitive preliminary model. For convenience the input to the model is the desired flap position as a relative value. The simulation results are obviously not rich in content because the model does not do much, especially it does not implement any synchronization. However, as the flap panel itself has not yet been properly modelled, the flap positions do not differ anyway.

The functionality is only very rudimentary and therefore the specifics of the several nodes are not shown here.

2.3.2 The virtual prototype

The virtual prototype comprises three levels with different kinds of information:

1. The first level contains the global cluster information. That is the specification of the different subsystems and the TTP messages, incoming and outgoing. A TTP message may be produced by exactly one but consumed by multiple subsystems. This is shown in Figure 2.5.

   Additionally the hardware nodes of the cluster are specified in a special view (see Figure 2.6). To avoid the name clash between *SCADE node* and *TTP node*, in TTPlink the TTP nodes are simply called *hosts*. The system uses redundancy by using two drive units per flap panel. It consists of the known parts as described before: ACE, MCE, motor, CSB, PPU and mechanical parts. If one side fails, the other side will move the whole flap panel with half the torque. The requirements document explicitly separates the two units of one flap panel. This means each unit can be explicitly addressed by a sender and each unit can intercommunicate with its partner via TTP. Therefore the messages that both units send and receive can differ. So the *fault-tolerant communication layer* of TTP cannot be used and both units must be modelled as independent subsystems. This is reflected in the mapping of TTP nodes to

subsystems: Each subsystem gets implemented by exactly one node.

This level of information is enough to generate the Message Descriptor List (MeDL). The model will be exported into a TTP-Plan cluster database and TTP-Plan will generate the MeDL. Unfortunately this feature could not yet be tested because the toolchain lacks one linking part between TTPlink of Esterel Technologies and TTP-Plan: *TTP-SCADElink* is an additionally needed tool of TTTech and fills this gap. It could not yet be provided because TTTech implements some improvements at the moment, but it is announced to be provided in the near future (see Section 2.5 about details on the future work).

2. The second level contains task information. Each subsystem only includes tasks and local messages between tasks. Task properties such as task name and time budget are specified here.

   SCADE can export this information to TTP-Build which merges it with the cluster database in order to form the fault-tolerant communication level (which is not used here) and the task schedules of the nodes (TaDL).

   Just as the automatic MeDL generation, this feature could not yet be tested.

3. The third level comprises the functional behaviour of the tasks. The implementation is not as simple as in the purely functional model. The tasks have to be completely separated. No other data transfer between tasks is provided by SCADE instead of the TTP messages. So modelling of the environment beyond task borders is hardly possible. Our solution will be to model the environment with the mechanical properties of the flap panel in Matlab Simulink and import this model to SCADE. See Section 2.4 and 2.5 for further information.

2.4 Using Matlab Simulink models as black boxes in SCADE

The mechanical part of the high-lift flap system interacts with the environment directly and therefore its behaviour follows very complex rules. Though it seems easy to model its logical functionality, there is a great effort to be done to simulate its behaviour within the "real world". An ideal flap would only have three states: *opening*, *closing* and *stopped*. The flap would have some absolute position, which increases in the *opening* state, decreases in the *closing* state and stays the same in the *stopped* state. Maybe there would be some speed level

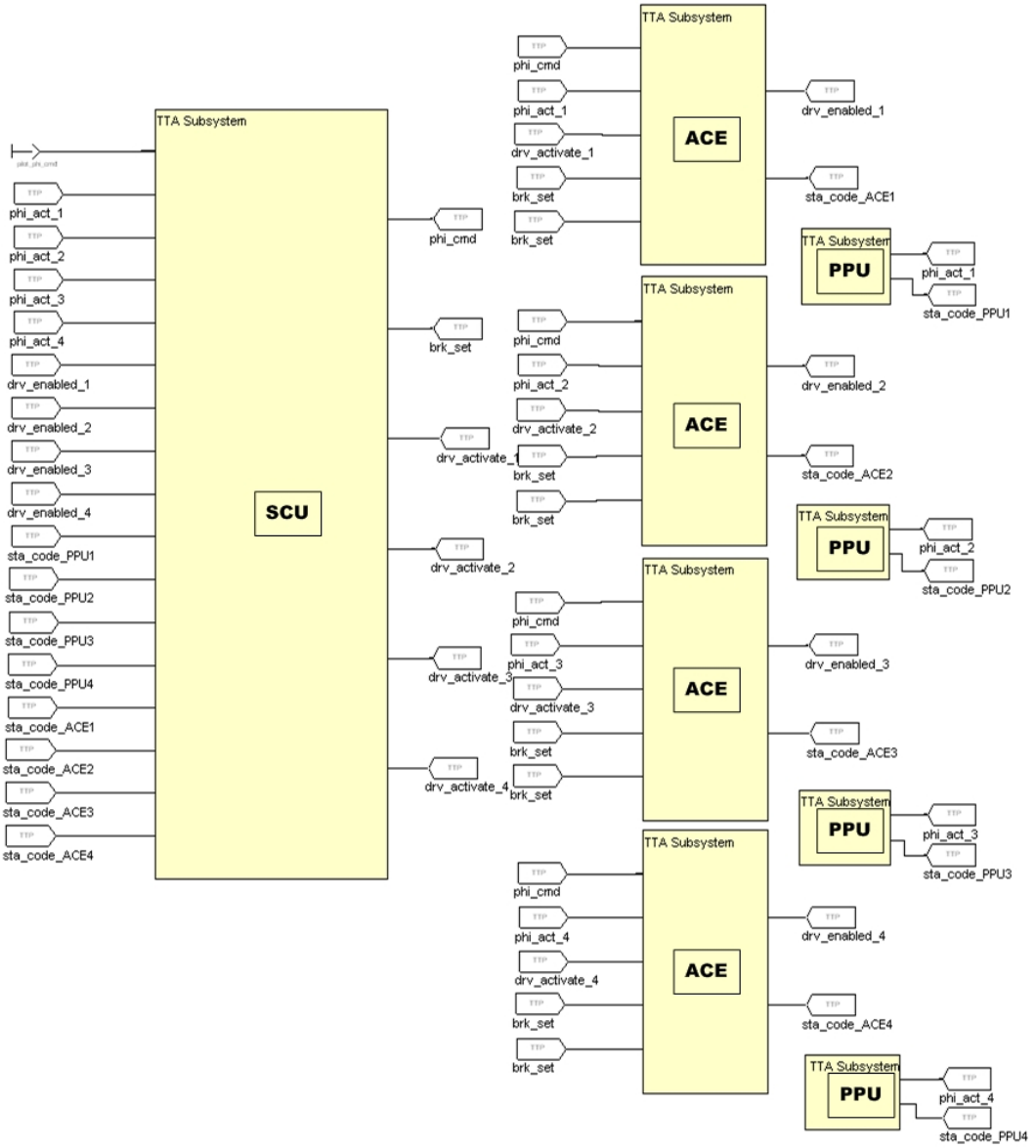


Figure 2.5: The subsystem level of the entire flap system. Subsystems and messages get specified.

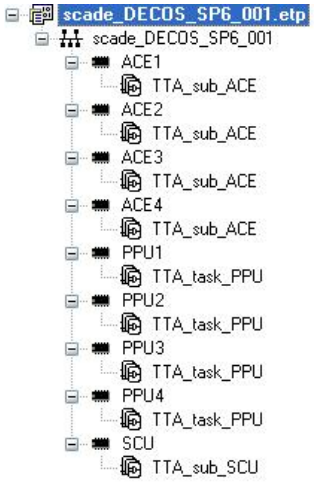


Figure 2.6: Mapping of TTP nodes to subsystems. The flap system does not use the fault-tolerant communication layer: No subsystem is mapped to multiple nodes.

which models the moving speed of the flap. Obviously this level of abstraction does not do justice to the interaction of this complex application. It simply ignores effects such as friction, slackness or slipping. The lack of perfection in the system manifest itself twofold: On the one hand the absolute position cannot be measured accurately and on the other hand the position can be influenced by more elements than the flap motors (which are not ideal, either).

To test if the possible faults are covered by the safety features of the model, one needs a simulation of the mechanical part of the flap that takes all these considerations into account. To get a virtual flap that is as close to reality as possible, one needs a very complex model regarding the non ideal environment. The *Technical University Hamburg-Harburg* (*TUHH*) has built such a model with great detail in several former projects in Matlab Simulink. It uses many highly non-linear calculations and simulates the behaviour of the physical existent test bench quite accurately. Figure 2.7 gives some indication of the complexity of this model of the flap panel. It shows a screenshot of only the first level and there are at least two deeper levels in the different components.

As the model, which covers the MCE and the mechanical parts of the flap, was developed in other projects, its source code will not be freely available for the DECOS project. Ultimately this is not a problem, as not the model itself is needed, but only its behaviour. To check the system behaviour, the model only needs to know *how* the flap would behave, but not *why*. So some kind of *black box* would be sufficient to be used in our model. Therefore we have built a binary file from the Matlab Simulink model that can be imported as

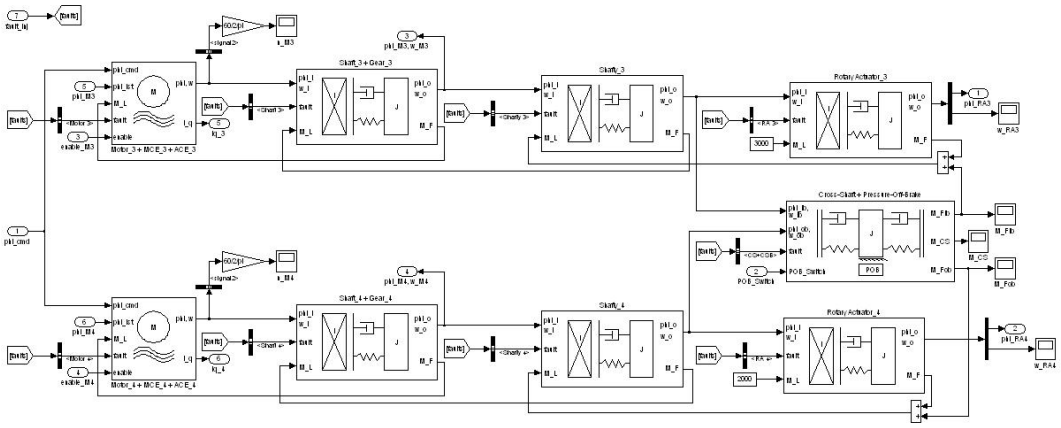


Figure 2.7: Screenshot of the first level of the preliminary mechanical, hydraulic model of the flap panel developed by TUHH indicating its complexity (that you cannot recognize anything is meant to be ☺). Source: TUHH

an external node into SCADE. The node has the same inputs as the Simulink model does and produces the same outputs, but the functionality is disguised to the user within the binary.

2.4.1 Building a Dynamic Link Library of a Matlab Simulink Model

Building a binary that executes the simulation of the Matlab Simulink model is straighforward. The result of this seamless process is an executable that runs the complete simulation:

1. The user models the system in Simulink, and Stateflow if applicable.
2. The Realtime Workshop builds C code from the model blocks with all needed header files.
3. A compiler builds a binary of these C-files for the desired platform: in our case for a Windows PC.
4. The result can be executed and it simulates the model with the simulation settings set within Matlab Simulink.

But that is not what we want. We do not need to run the whole simulation, which includes the setting of input variables for the whole simulation and which runs all simulation steps successively. What we need is an interface

which allows us to set the input variables for each single simulation step and then allows us to execute such a single step and to return the output results. The seamless build process of Matlab Simulink with the Realtime Workshop and the Realtime Workshop Embedded Coder does not support the building of such a binary that offers the interface we need. One has to do some manual adaptations to the code that the Realtime Workshop produces.

Thus we need to manipulate this process manually. What we need is the C code that offers the desired interface and a compiler which allows us to build a binary library out of the code in such a way that its functionality can be called from within other C-programs.

The Realtime Workshop Embedded Coder produces a code that is well readable, so we use it to produce the C code. It provides procedures for initialization and termination of the simulation, access to the inputs and outputs and a procedure to advance the simulation one step. For compilation we use Microsoft's Visual Studio 6, because it is the compiler which is most widely used by the relevant project partners. It supports the building of a *Dynamic Link Library* (*DLL*) for Windows. Dynamically linked means: the library file does not need to be (statically) linked to the external C-program when the C-program is built. It keeps a reference to the produced DLL file and will call the procedures from the DLL file rather than from its own binary, as it would happen if a static library were linked into an executable. This allows the modeller of the Simulink model to change the model and build a new DLL without the need of a recompilation of the other C-program which uses this library, unless the called interface is changed. Whenever a small change of the model is done, the user of the library simply copies the new DLL file but does not need to recompile his or her own programs.

If one wants to build a DLL file, one has to fix which functionality will be available to the outside world. Specifically *export* declarations to each procedure description must be added to the header file of the source code. Access to variables is not obvious, as a DLL can be used by more than only one caller. So one has to declare whether the memory of these variables will be shared between the callers or whether each caller gets its own instances of them. To ease things, we add *get* procedures for the inputs and outputs that can be easily made available to the DLL interface.

So the build process of the DLL file looks like the following:

1. The user models the system in Simulink, and Stateflow if applicable.

2. The Realtime Workshop with the Realtime Workshop Embedded Coder builds C code from the model blocks with all needed header files for the desired subsystem.

3. The user manually adds the *get* procedures for the inputs and outputs to the main C file of the model.

4. The user adds the declarations of these *get* procedures to the main header file.

5. The user adds the *export* declarations to all required procedures in the main header file.

6. Visual Studio builds the DLL.

The result is the following:

- A *DLL* file which includes the binary information about the main functionality.

- A *LIB* file which must be explicitly linked to the C-Program that uses the library. This static library file contains the link to the dynamic library file.

- The inclusion of the main *H* header file is needed for the caller of the library to let his C compiler know about the interface of the library.

Doing the steps 3, 4 and 5 manually is a significant effort and error prone, especially for an unexperienced developer who does not know the particulars of C code, compilers and the building of DLL files. So we developed a command line Java application that applies the needed changes in the source and header files automatically. With this little helper and a more detailed instruction map (which we made available) an unexperienced user is able to build the Dynamic Link Library from the Simulink model.

As the Realtime Workshop puts some functionality into the main header file, the file might offer too much information about the model to the user and the black box character might be violated because the user of the library needs the header file declarations. So the Java application automatically builds a new header file that only includes the needed declarations but no details about the model.

Another advantage of making the changes automatically by the macro is the following: The modeller can change the Simulink model and then build the library with only a few mouse clicks, once the Visual Studio Project is set up correctly. So changes in the model can be easily fed into the library.

Due to an acknowledged bug in the Realtime Workshop and the Realtime Workshop Embedded coder (which was discovered by this work) the input and output variables of the C code have sometimes not the same names as the

corresponding input ports and output ports in the Simulink model. So it is nearly impossible to use the library for a user who does not know about the particulars of the model. The correct use of the inputs and outputs becomes a guessing game. Therefore we manually added macros to the main header file that grant access to the inputs and outputs by using identifiers.

2.4.2 Using MS Windows DLLs as Imported Operators in SCADE

The DLL of the Simulink model offers an interface to call the following functionality:

- Initialization of the simulation.
- Setting input values, getting output values. For each value a single macro is available, hence the values can be modified individually instead of all together in a single procedure.
- Execution of one simulation step for the whole Simulink model.
- Termination of the simulation.

Our SCADE model is meant to include the PPUs and the ACEs as single nodes, because keeping the mechanically connected components together in one node does not represent the architecture properly and TTA subsystems must comprise independent units.

An *imported node* in SCADE can be used to import functionality to a SCADE model that is only available as C code. Hence imported nodes comprise C code which implement their functionality. Therefore it is possible to write a C file for each PPU and for each ACE. In such a C file the functionality of the imported operator must be implemented and the code of each file will be called each time the operator is used in SCADE:

1. SCADE passes the inputs of the operator to the code (which is implemented as a procedure).
2. These input values are passed to the dynamic link library.
3. The outputs are read from the library.
4. The outputs are passed back to SCADE.

The main problem is that the dynamic link library only possesses one global step function. So calling this function in each of the four imported operators in SCADE would proceed the simulation of the Simulink model four steps and not only one.

An alternative would be to use four independent instances of the dynamic link library, one instance for each imported SCADE node. This would mean that the other operators in the library instances are fed with dummy input values only. However, this approach would not work either, because it ignores the fact that the single ACEs and PPUs are linked by the mechanical parts and therefore are *not* independent. Hence the global step function of the library is inherently needed, since it does not only advance each component one step but the whole mechanical system as well.

We use a double buffer strategy to tackle this problem: In each code of the imported operator, the inputs of the current step are passed to the dynamic link library and the outputs are read by it and passed back to SCADE. A global counter counts how many imported operator procedures have been already called. If all imported nodes have processed their inputs and outputs, the global step function of the dynamic link library is called and the Simulink model simulation advances one step.

This means: the simulation is fed by the current input values, but the outputs read from it are the outputs of the last step. Thus the SCADE model works with output values that are one step behind. We assess this introduction of an additional delay in the data flow as justifiable, because it hardly carries weight: The TTA triggers the participating components once in each TDMA round, which is one millisecond. The mechanical model must be simulated with a step size of 10^{-4} s and its step function is therefore called ten times faster and thus the additional lag affects the SCADE model only every tenth step of the simulation of the Simulink model and introduces a delay of the mechanical reactions of only 0.1 ms. As this delay only covers the mechanical reactions (that is to say the PPU output values), it can be neglected.

This way it is possible to import several nodes from only one Simulink model DLL and to run them properly. We tested the correct behaviour in a little test SCADE model with one ACE and one PPU (see Figure 2.8).

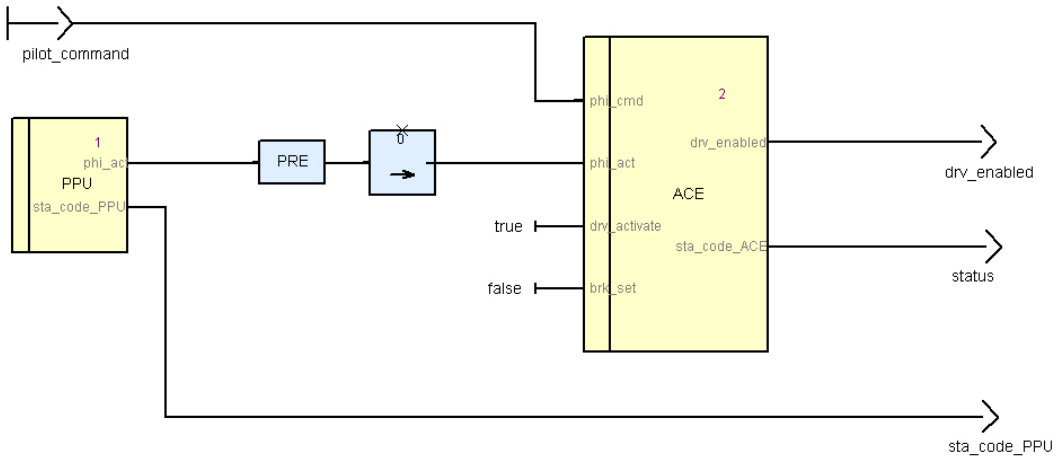


Figure 2.8: Integration of ACE and PPU into a SCADE model as imported operators, both from the same Windows DLL

2.5 Future work

2.5.1 Full integration of the Simulink model into the SCADE model

The imported operators have not yet been integrated into the SCADE model that models the TTA according to the requirements [Koc04]. This is because the input and output types of the two models have not yet fed together. The Simulink model modelled by the TUHH is only a preliminary model without correctly typed inputs and outputs. All values are typed as real values and therefore are implemented as double numbers in the C code. Even the boolean flags as `brk_set` and `drv_enabled` are implemented as real values. But the SCADE TTA messages require correct typing and it would be desirable to stick to the correct typing. Conversions of the types would be needed at each input and output signal of the imported operators, either in C in the imported node code Sections or in SCADE before the inputs resp. behind the outputs.

As the TUHH planned to change the homogeneous real typed variables to the correct types in the future, the final integration of the Simulink model will be done then in order to avoid the tentative conversion functions.

The Simulink model of the TUHH only models the left flap panel. Hence we will need to integrate two instances of the dynamic link library to the model. This will be tackled when the integration of the Simulink model into the SCADE model is possible as described above.

2.5.2 Generation of the Message Descriptor List (MeDL)

One of the key results of our model will be the generation of the TTP communication schedule (Message Descriptor List - MeDL). SCADE is used to specify the TTA and an interface to the TTTech Tools [TTT] allows to use the cluster compiler of TTPplan to generate the MeDL automatically. To run this process seamlessly you need:

- SCADE version 5.0 (by Esterel Technologies)
- SCADE TTPlink (by Esterel Technologies)
- TTP Tools TTP-Plan, TTP-Build (by TTTech)
- TTP-SCADElink (by TTTech)

As we already have the first three products, we can build the model, but with the lack of the TTP-SCADElink tool of TTTech, the automatic transfer of the model data into the TTTech tools is not yet possible. TTTech announced that we will get TTP-SCADElink at the end of February 2005, since they still modify the product.

Then the MeDL can be generated, but the requirements [Koc04] do not reveal all needed TTP messages on the bus. So the final schedule will be generated when more information about the messages is available (which will be given in the present specification phase).

The MeDL is a global data structure with great impact for every local component. Preliminary schedules are of little use, since new messages usually change the timing of the complete schedule.

2.5.3 Implementation of logical behavior

The synchronization strategy and fault reactions have been developed mainly by TUHH. A failure mode and effects analysis (FMEA) has been employed by TUHH to reveal possible failure modes. If there is any more specific information about the logical behaviour available, we will try to implement this behaviour in the SCADE model.

2.5.4 Detach the ACE from the Simulink black box

At this point of work the scope of the Simulink model by TUHH covers the following:

- The mechanical parts of the flap panel.

- The cross shaft brake (CSB).
- The Position Pickoff Units (PPU).
- The Motor Control Electronics (MCE) and the electrical motors.
- The Actuator Control Electronics (ACE).

The first four items are highly dependent on the mechanical properties of the installation and are therefore needed to model the mechanical subsystem correctly. The MCE includes no logical fault prevention behaviour but detailed motor property dependent control, and it does not communicate via the TTA bus.

However, the ACE will likely include a significant amount of logical behaviour as it builds an inner control loop for the positioning of the flap panel with the PPU [Koc04, Gei04]. As the ACEs are main participants in the time-triggered communication via the TTA bus, we need to model the communication parts of the ACE within our SCADE model. But as the ACEs include a high amount of control modelling developed by the TUHH, they will be modelled by the TUHH as well.

An enrichment to our SCADE model would be the detachment of the ACE of the black box library. The ACE is being developed within the DECOS project and therefore its model could be made available to all project partners. We suggested that TUHH would provide us with a white-box version of the ACE, which is the Matlab Simulink model itself. Esterel Technologies delivers gateways for Simulink and Stateflow models into SCADE, which might allow an integration of the ACE with all its functionality. Simulink blocks will be automatically translated into SCADE nodes and Stateflow state machines will be translated syntactically into Esterel's Safe State Machines (SSM). With this approach there will be a SCADE model of the full functionality of the ACE available, including the communication part and the control part.

This enables us to

- generate code for the ACE,
- extend any verification processes to the full ACE,
- have clean interfaces as they exist in the real test bench (the interface will be the one to the MCE rather than across the ACE).

TUHH signalized to cooperate in this point and therefore this topic can hopefully be tackled in the near future.

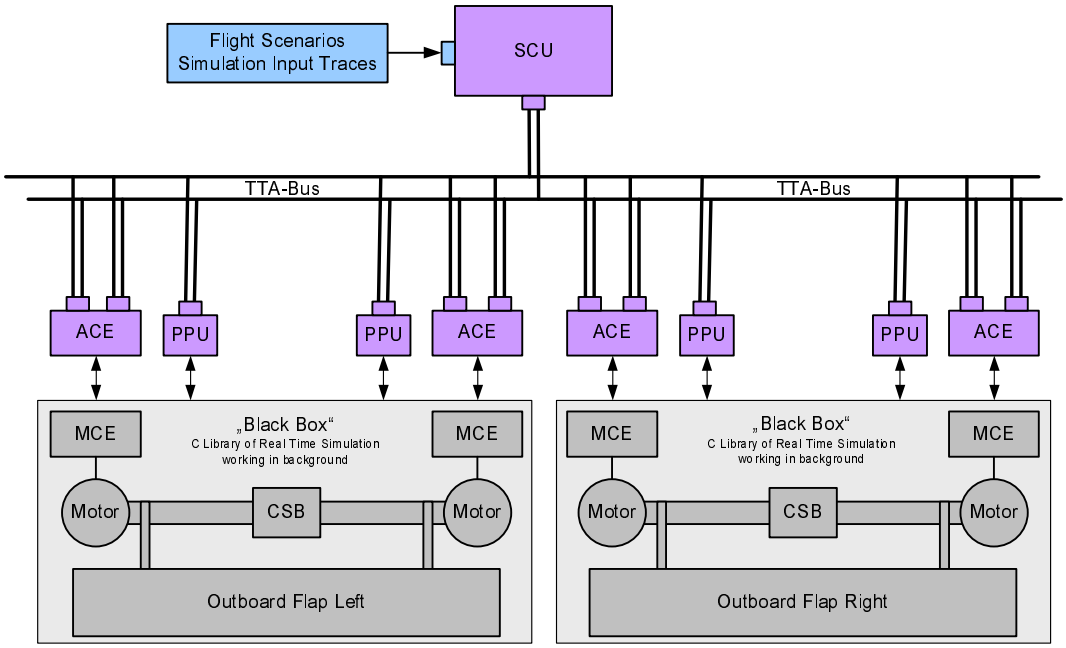


Figure 2.9: Possible architecture of the model: ACE, PPU available as white boxes and the mechanical subsystem works in background as black box in the C library

As the translation processes of the Simulink to SCADE Gateway and the Stateflow to SCADE Gateway are not perfect (State machines are simply translated syntactically), manual revision is necessary. It must be deliberated about the manual adapting efforts needed to assess whether the approach is feasible or not. This can be done after the Simulink and Stateflow models are provided to us and reveal their model properties that make manual adoptions necessary.

It can be discussed whether this approach is feasible for the PPUs as well.

Figure 2.9 shows how the architecture of the SCADE model could look like.

3 System Response Requirements in a TTA

3.1 End-to-end latency

End-to-end latency is critical in hard real-time systems. The developer has to ensure that the system response requirements are fulfilled. Using TTP might give the impression that this problem is inherently solved. However, message latencies in TTP did not disappear, they simply got fixed. Different control loop strategies and especially bad scheduling can introduce latencies that a naïve user would not expect. Therefore this topic needs to be analysed. Since the timing properties of TTP are quite complex, it should be assessed how modelling tools can assist the developer in fulfilling the system response requirements.

3.1.1 Latency in general

Looking at a hard real-time system [Kop97], *e.g.*, the high-lift system we are modelling, the main focus is put on the response requirements of the system. Hard real-time requirements imply that there are tasks to do that must be executed within a defined time for completion. In our case we have a reactive system and build a close control loop to keep the system under control. An event in either the environment or other system components needs a reaction of the system within a well defined time (see Figure 3.1), otherwise reactions could occur so late that the system gets out of control.

Think of an *airbag inflation system* in a car for example: A crash can happen at any point in time. The system has some response requirement that says: The airbag has to be fully inflated $x\ ms$ after the first contact with the obstacle. Otherwise the head of the car occupants will likely have thudded before the airbag could keep them to the seats. This time duration of $x\ ms$ depends on many factors. There are especially the driving speed and the construction of the car that determine the period of time that passes until the inflation of the airbag will be too late. If you assume a maximum speed of the car, you will get a minimum *latency* requirement of the system only by the physical attributes

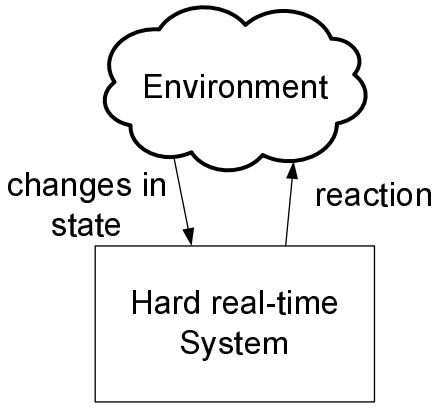


Figure 3.1: A hard real-time system interacts with the environment and needs to produce reactions to events within a special amount of time

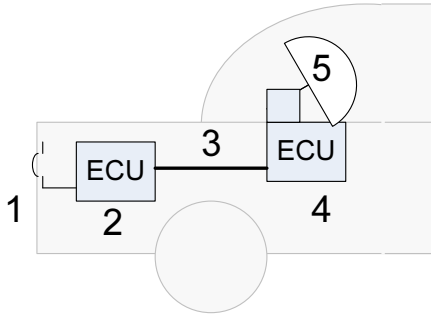


Figure 3.2: The elements of latency in a simple control application: 1. sensors, 2. computing of sensor ECU, 3. duration of communication, 4. computing of actuator ECU, 5. physical delay from airbag ignition

of the environment.

To fulfil the timing requirements, you have to tackle each part of the hard real-time system that participates in producing the overall delay. So we decompose the system in order to identify these elements. Assume a control application with only one sensor unit and one actuator. The sensor unit sends its sensor values to the actuator unit which acts, if the values indicate a need for reaction. For the airbag example this is shown in Figure 3.2 in a schematic view:

1. If some state changes in the environment which requires a reaction, the first part of delay is introduced by *the sensors* of the system. For example temperature sensors indicate a changed temperature very slowly. In the airbag example this would be the delay that is introduced by the sensors that have to detect a collision.

2. The *electronic control unit* (*ECU*) that is connected to the sensors is the next step in the flow of information. Any computations that must be done, before a notification can be sent to other components, introduce another delay. In the airbag example this would be the controller that is connected to the sensors in the front of the car.

3. The information must be distributed to the components that receive the information. The *message delay* is the time between the point in time where the sender provides the information and the point in time where the receiver can use it. This includes the physical delay of transmission when one sends the message and any additional communication protocol overhead that introduces an additional delay. The sensor controller of the car has to send a message to the airbag controller in the car cabin.

4. The receiver of the message has to analyse the information. Any *computations* that have to be done until the final decision about some reaction creates another delay. The airbag controller in the cabin might need some time to process the message and to decide whether it releases the airbag or not.

5. If a reaction is commanded by the receiving controller, there may pass some time from this command till the point in time when the reaction has completed in the environment. This delay in the airbag example is created by the airbag itself: When the airbag controller commands the airbag to inflate, it ignites a chemical reaction that produces a certain amount of gas (*e.g.*, nitrogen) in a very short time which inflates the airbag. The time from command to full inflation is this delay.

If the airbag exceeds the allowed latency, this could obviously do serious harm: If it does not ignite at all, the catching function of the airbag does not go into operation and therefore the passengers are likely to get hurt. If, however, the airbag ignites *after* the crash has already pushed the passengers to the front, the sudden backstroke could do additional harm, *e.g.*, break the passenger's neck. This shows that latency must not be too long under any circumstances: Having no airbag might be better than having an airbag that ignites too late.

The delays of 1 and 5 are given by the physical properties of sensors and actuators. These are given by physical properties and cannot be changed (at least not by computer scientists). Thus most relevant parts of the delay are 2, 3 and 4, *i.e.*, the duration of any computations within ECUs and the message transfer.

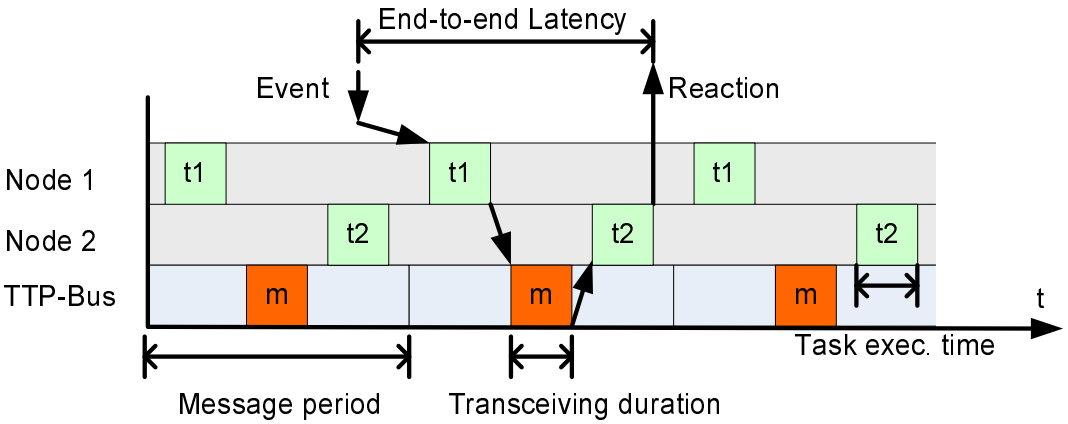


Figure 3.3: Latency under usual conditions in a TTA

3.1.2 Latency in the Time-Triggered Architecture

We want to disregard the physical parts of the latency (sensors and actuators) and only look at the delay of the computing tasks at the ECUs and the message transfer.

To keep the overview over the terms and abbreviations, review Appendix A at page 89, whereas the concepts will be introduced piece by piece in the following sections.

In an event-triggered communication there is no guaranteed maximum message latency: By collisions and other messages on the bus the message transfer can be delayed for an arbitrary amount of time if not special features, such as bit arbitration in CAN [ISO93], give priority to safety-critical messages.

See Figure 3.3 for the factors that influence latency in a usual TTP application. It shows a simple control loop with one sensor and one actuator node like the airbag example. Our *ACE-PPU-control-loop* in the aerospace demonstrator will be such a subsystem, too.

As shown in Section 1.2 the system works like this:

Node 1 is the sensor node that regularly executes a task, *t1*, that reads the sensor values. We assume that the value is read at the beginning of the task execution and therefore only events will be detected that occur until the start of the task.

If an event occurs, *i.e.*, a change of state in the environment, the task will be notified by the sensors. Then the task processes this piece of information and stores a message onto the *communication network interface* (*CNI*) of its TTP network controller.

At the next TDMA slot of the node the message will be transferred. After final transmission, the TTP controller of *node 2*, the actuator node, has stored

the message in its CNI.

A task at the host computer of node 2 regularly reads the messages and processes its content. If the message indicates the need of a reaction, the task will trigger some reaction, *e.g.*, the ignition of the airbag resp. a changed motor speed at the flap panel motors.

Therefore *end-to-end latency* in this application comprises the following parts:

1. An event can happen at any arbitrary point in time, but the task of the sensor node is regularly run with a fixed period. Therefore some time passes between the change of state and the beginning of procession by task *t1*.

2. The *execution time* of the task *t1* is another part of the delay. We assume that the message is passed to the CNI at end of the execution of the task and therefore nearly the whole task execution time is part of the end-to-end latency. An upper limit of this time is the *worst-case execution time* (*WCET*) of the task which manifests as the granted *time budget* in the TTA. Analysing the WCET of a task is a topic of its own, we will assess it in Section 3.1.3. As we are interested in an upper limit of the end-to-end latency, we assume a fixed execution time, which is the WCET of the task.

3. Next part is the *pre-send duration* (*PSD*) of the message, *i.e.*, the time between the end of the task when the message has been written to the CNI and the actual transmission which is triggered by the slot in the Message Descriptor List (MeDL) of the TTP communication controller.

4. The *slot time* of *node 1* is the next part. It is the time in which the message is transferred via the TTP bus and provided to the CNI of the receiver. The time until the message can be read by the receiver depends on properties of the TTP controller, which are the periods of time that the controller needs for local processing of the message (*e.g.*, time for access to the CNI, processing the CRC, etc.). These periods depend on the controller type and are relatively short (at our AS8202NF controllers the post receive processing takes 11 μs). We denote the whole slot time including all pre and post processing phases and the inter-frame gaps as the *transceiving duration*.

5. The time between providing the message to the CNI and reading it by a task is the *post-receive duration* (*PRD*).

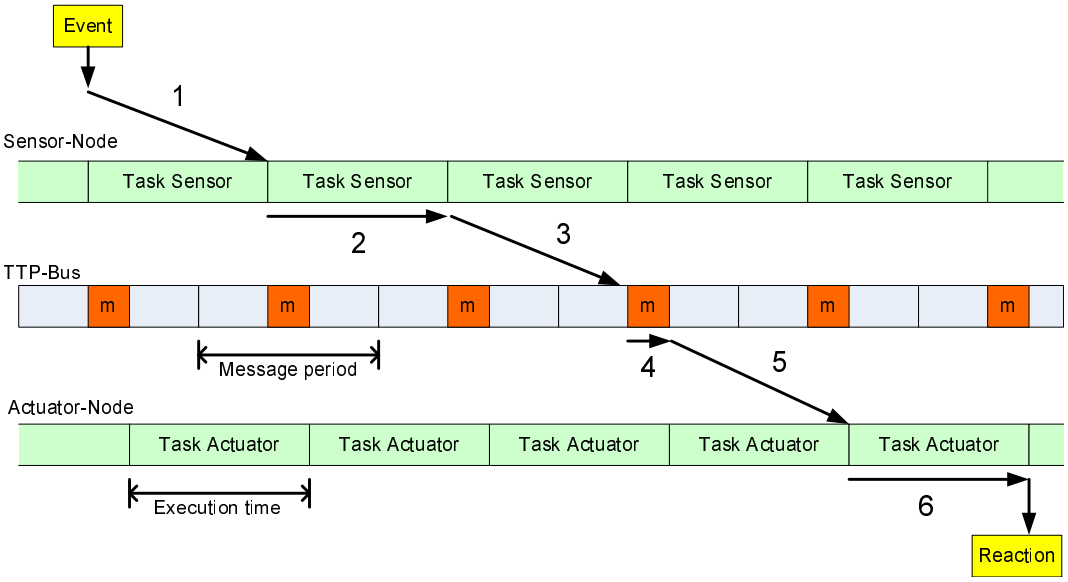


Figure 3.4: Worst-case latency in a simple control loop

6. The execution time of the task *t2* is another part of the latency, since the received message must be processed in order to state a decision about a reaction.

3.1.2.1 Worst-case latency

The end-to-end latency might be relative good, if the WCETs of the tasks are short and the tasks are well scheduled. This might introduce a latency of less than one message period extended only by the fact that the event itself can occur at any arbitrary time between two task executions.

But a worst-case latency in this simple control loop of only one sender and one receiver can be relatively long, if the WCETs of the tasks are long and the tasks are disadvantageously scheduled. This worst-case is depicted in Figure 3.4:

Call the message period T_m. This can be the round time of the TDMA round, if the message is sent in every round, or some multiple of it, *e.g.*, twice the round time, if the message is sent only every second TDMA round. In order to be able to send all messages and to receive all messages, there is $t1_{WCET} < T_m$ and $t2_{WCET} < T_m$. Assume the execution time to be nearly T_m but to be able to let the operating system tasks write and read the messages to and from the CNI between the task executions. Schedule the tasks very badly, which means: The sending task finishes its execution directly *after* the TTP controller has begun the transmission. That means this message will be transferred in the

next sending slot. The receiving task reads the CNI directly before the new message has been provided to the CNI, so it reads the message of the past slot. This introduces an end-to-end latency as shown in Figure 3.4:

1. Assume an event occurs directly *after* the sensor task has read the sensor values. Hence the change of state of the system will be noticed by the next task execution.

2. The task processes the value and needs its long WCET for it.

3. As the task is scheduled badly, the TTP controller accepts the message only for the next sending slot. Therefore the *pre-send phase* is nearly T_m.

4. The message gets transmitted via the TTP bus. This introduces the slot time as a part of the delay. In a worst case this *could* be as long as the message period, but this would mean that the message is the only one on the bus and its content is relatively big resp. the message period is very short. This is very unlikely so we assume that we use a simple message, *e.g.*, a single integer or float value.

5. Just like the pre-send phase, the *post-receive phase* becomes very long because the receiver task is that badly scheduled that it cannot use the actual message but the message of the past slot.

6. The WCET of the receiving task will pass until finally the reaction is triggered.

1 introduces a delay of T_m. This cannot be avoided as the point in time when the event occurs cannot be influenced, so the time between the reading of the sensors will always be part of the worst-case latency. As $t_{WCET} \approx T_m$, 2 and 6 introduce a delay of T_m each. The bad scheduling of both tasks adds a maximum delay of another T_m each. This results in a worst-case latency WCL:

$$WCL = 5\ T_m\ + \Delta m$$

where Δm is the message transfer duration. If we assume that Δm is relatively short and the WCETs of the tasks have to be smaller that T_m because the operating system of the nodes needs some time for administration, it is roughly

$$WCL = 5\ T_m\ .$$

This example is obviously very artificial and is unlikely to occur this way in a real system. But it shows that an end-to-end latency of much more than one message period is possible by bad scheduling and long WCET.

This was an example of just a simple control loop with one sender and one receiver. Consider a control application, where the flow of information includes more nodes in the cluster. As shown the *scheduling* of the tasks is critical for the WCL. The flow of information has to be taken into account and the WCETs of the tasks need to be provided as exactly as possible in order to help the schedulers to optimize the schedules.

3.1.3 Worst-case execution time

Worst-case execution time is an active research topic, because it is very difficult to conquer [MML97, EES01]. Current computers are designed for good average performance and therefore introduce sophisticated dynamic techniques to improve it, such as pipelining, caching, out-of-order execution, *etc.*. These dynamic components impose serious difficulties on static analysis of the execution time of programs. Therefore the statement "this airbag controller will compute at most 17.5 *ms*" is very hard to prove.

This work does not cover worst-case execution time analysis itself. This work tackles the question how modelling tools help the developer in calculating the end-to-end latency, and this includes the WCET as we have seen before.

The *TTTech* tools (see Section 1.3.2.1) do not offer any WCET methods of analysis by themselves. The WCET values have to be provided by the developers as so-called *time budgets*. These are used by the schedulers to generate the task schedule for the nodes. *SCADE* of *Esterel Technologies* (see section 1.3.2.3) adds no such feature to the design process. You simply provide the *time budget* of each task by inserting the values into the property forms of the SCADE user interface. This implies that the developer has identified the values by himself, by analysis and/or measurements.

Only *Matlink* of TTTech for *Simulink* (see Section 1.3.2.2) provides an additional feature, described in the following section.

3.1.3.1 Execution time measurements of Matlink

Matlink allows the developer to measure the execution time of the tasks automatically. The workflow is depicted in Figure 3.5:

1. The developer selects that he or she wants to measure the execution time for his tasks.

2. A TTP message that sends the measured execution time to the TTP bus is generated automatically. There is no receiver, because the message is only to be read by the developer.

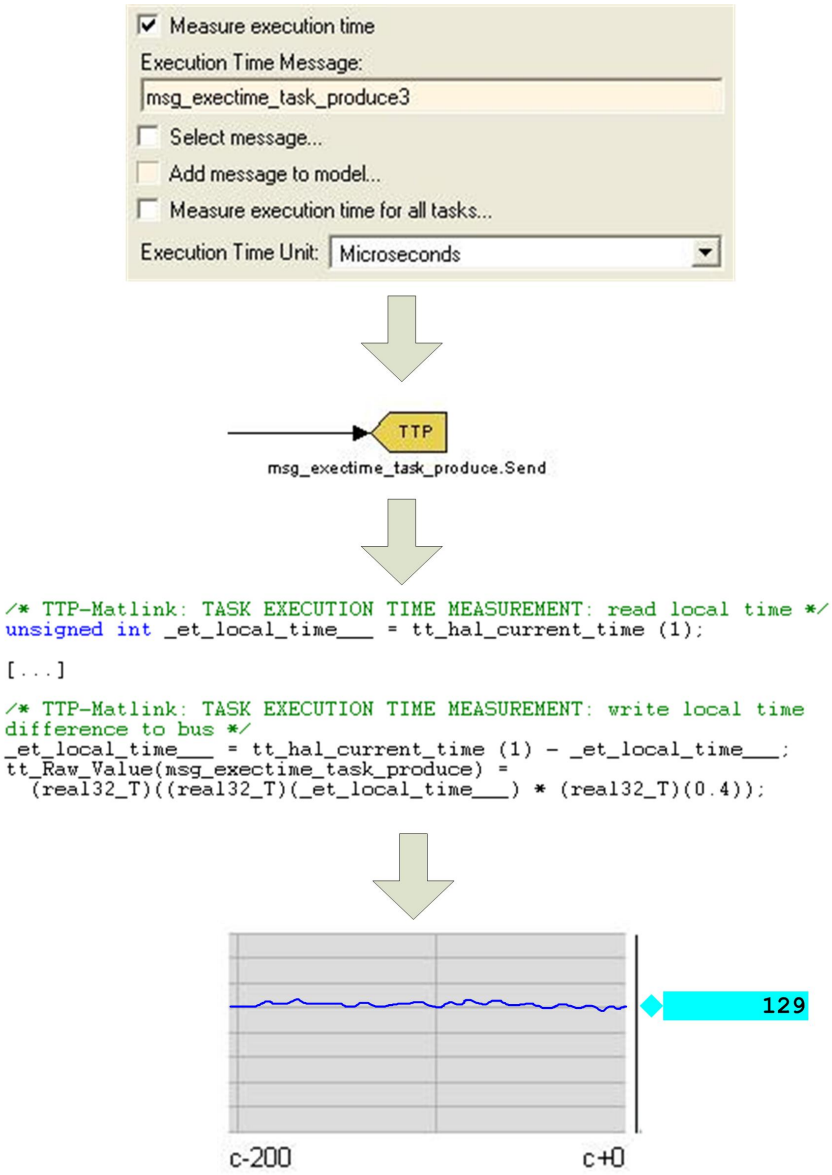


Figure 3.5: The WCET-measurement feature of Matlink

3. The automatic code generation adds the specific C statements for time measurement to the code of the task.

4. At execution time, the values can be read with the help of a *monitoring node* and *TTP-View* can display them to the developer.

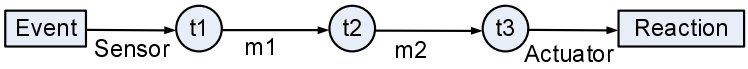
The developer could take the highest value he measured, add some safety margin to it and use this value as the *time budget* of the task.

Even if the developer knows that the longest path in the task has been executed, this method does not reveal the actual WCET of the task. Dynamic components of the hardware are not taken into account, *e.g.*, the cache hierarchy. So this method might show an average execution time which will be sufficient for common applications, but it does not discover the real WCET.

3.2 Event Recognition Latency vs. Information Flow Latency

Consider a brake-by-wire system as it is modelled in Figure 1.16 on page 29: a pedal subsystem is connected to a sensor which measures the current pedal position and sends this value to the TTP bus; a calculation subsystem fetches the pedal position and the wheel speed from the TTP bus and calculates an appropriate brake force; a brake actuator subsystem is connected to wheels by the mechanical brake and a wheel speed sensor. It uses the brake force value and ultimately brakes the wheels, while the current wheel speed is fed back to the TTP bus.

This example shows that control loops usually comprise more than only two nodes with one message between them. We will look at the following message flow:



The control loop comprises three nodes and each node executes a different task. Task $t1$ reads a sensor value and sends a message $m1$ to task $t2$. From $t2$ the message $m2$ is sent which is consumed by task $t3$. This is connected to an actuator that performs the appropriate reaction.

The end-to-end latency arises from all delays during the flow from event to reaction.

In order to analyse the end-to-end latency or to synthesize the system settings from a given maximum latency, we need to divide the end-to-end latency

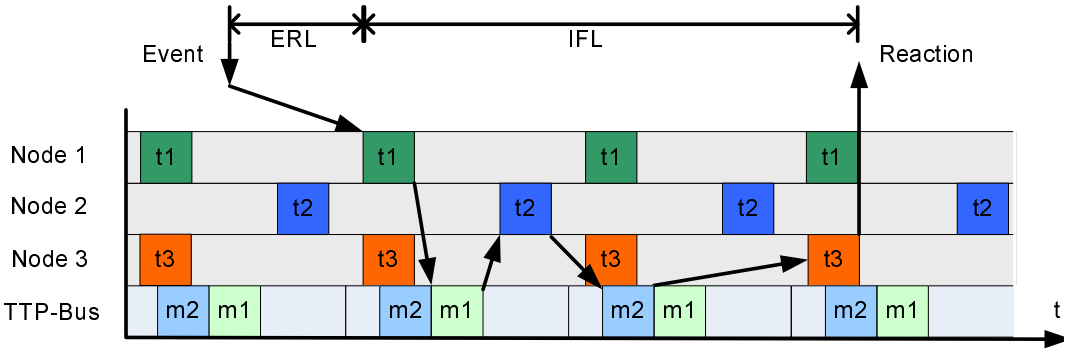


Figure 3.6: Event Recognition Latency (ERL) vs. Information Flow Latency (IFL)

into two parts, which are depicted in Figure 3.6 for this example and an exemplary schedule.

The *event recognition latency* (*ERL*) describes the time between an event and the recognition by the task of the appropriate sensor. In the time-triggered architecture the sensor tasks are regularly executed with a well defined period. Therefore the earliest point in time when an event resp. a change in state of the environment can be recognized is the next time the task is executed. Due to the regular execution of the task, the ERL is no exact value, because it can vary: if the event occurs a very short time before the task execution, the ERL will be very short, too. If the event occurs directly after the task has been started and has read the sensor, the event only will be noticed by the next execution of the task. Hence the ERL will be nearly the period of the task execution. This effect introduces an undesirable but inevitable jitter. Therefore we only consider the worst-case ERL for response time analysis. This is the sensor task period.

The *information flow latency* (*IFL*) is the remaining time between recognition of the event and the reaction, *i.e.*, the time the information needs to flow through the system until the reaction is initiated. As communication in the time-triggered architecture is deterministic, this duration is fixed for any message flow in the system (this only applies for the TTA, not for a system which uses TTP for communication, but hosts that use other scheduling methods). This latency consists of the parts introduced in Section 3.1.2 for every transmitted message: The *execution time* of all three tasks, the *message send and receive duration* of both messages and the *pre-send duration* (*PSD*) and the *post-receive duration* (*PRD*) of the messages.

3.3 TTA System Development

3.3.1 Workflow and features of the current modelling process

The TTA System Development process has already been described in Section 1.3.2.1 about the TTTech tools and the interface to the modelling suites in Section 1.3.2.2 resp. 1.3.2.3. In this section we want to look at some more details according to the question on response time.

As mentioned before, the development process is divided into two fields of responsibility: the *system integrater* and the *subsystem supplier*. This dichotomy can be found in the tools of TTTech as well: *TTP-Plan* is used to specify the global cluster information, while *TTP-Build* configures the nodes individually.

Both tools generate schedules: TTP-Plan builds the global communication schedule, the MeDL, and TTP-Build generates the local task schedules for a node, the TaDL.

Each tool needs information about the system provided by the developer. This information can be imported from a model created by one of the modelling suites. This would only be mandatory information that is absolutely needed to generate a schedule. This is due to the assumption that the modelling suites are used for rapid prototyping only and not for final fine-tuning of the system.

The system databases of TTP-Plan (cluster database) and TTP-Build (node database) can be enriched by a variety of optional settings via the user interfaces of the tools themselves. For example, these cover a `max_membership_value` to adjust the fail silent behaviour and a `fixed_round_number` to fix the TDMA rounds in a cluster cycle to the `bcet`, which is the best-case execution time of a task for optimizations of the task schedules. The following describes the mandatory and optional relations and attributes that affect the response time most.

Basic Information Basic information comprises the cluster layout, *i.e.*, the number of nodes, the messages each node sends and the relation between messages and receivers, the *time budget* of each task and the global *transmission speed* of the physical layer.

TDMA round period The `tr_period` attribute globally sets the period of the TDMA round. As a full cluster cycle usually comprises more than one TDMA round and a node does not need to have a time slot in each TDMA round, the message period of one message can be a multiple of the TDMA round period. Therefore the TDMA round period is not that significant for

end-to-end latency.

Message Period The `d_period` attribute sets the maximum period for a TTP message. This is just a maximum value, because the actual message period has to be a power-of-two multiple to the TDMA round period. Hence any improper values will automatically be reduced to a message period `a_period` that is a power-of-two multiple of the TDMA round period.

PSD and PRD For each message the pre-send duration and the post-receive duration can be set to maximum with the attributes `max_psd` and `max_prd`. Thus the scheduler of TTP-Build is restricted to only build such schedules in which sending and receiving tasks are scheduled that way that the PSD and PRD are met by all corresponding tasks.

Message depends on Message One can define dependencies between messages with the association `Message_depends_on_Message`, *i.e.*, if message $m1$ is needed to generate message $m2$. This can be used to provide knowledge about the flow of information. To be declared dependent, $m1$ and $m2$ must apparently have the same period. `Message_depends_on_Message` tells the scheduler the order of the application tasks. With the `generation_lag` attribute one can define the allowed delay between the reading of a message and the sending of the corresponding output message. The value is no direct time span but it describes how many output messages may lie between the input and the output.

The value means:

0: as little delay as possible; $m1$ produces the earliest possible $m2$;

1: one message generation allowed between $m1$ and $m2$;

2: two message generations allowed between $m1$ and $m2$;

3: etc.

A value of 0 means that after reading $m1$ the answer must be written to the next occurrence of $m2$. In our example in Figure 3.6 this is the case for task $t2$. The value may not be bigger than the amount of corresponding messages in a cluster cycle.

3.3.2 Further options

The response time is not explicitly taken into account by the TTTech tools; neither for synthesizing the schedules nor for analysing the resultant schedules.

We want to discuss if it is possible to add new features that the tools would be able to handle end-to-end latency as desired.

3.3.2.1 Validation of System Requirements

Validation of the system requirements sounds simple: Take all resultant schedules and follow the flow of information. The actual latency can be computed by the formula

$$IFL = \sum_{i} ti_{WCET} + \sum_{j} (\Delta mj + PSD_{mj} + PRD_{mj}) \tag{3.1}$$

where i counts the tasks between the messages, j the messages in the information flow itself and Δmj the duration of sending the message, *i.e.*, the corresponding slot time.

Unfortunately the dichotomy of TTP-Plan and TTP-Build thwarts this simple approach: The *system integrator* that uses TTP-Plan to generate the global configuration data such as the MeDL is interested in the response time of the system. To calculate this, the global information is necessary as well as all local information of the nodes within the message flow that needs to be analysed. The *system integrator* is likely to receive from its *subsystem suppliers* an electronic control unit with binary software on it. It is unlikely that the suppliers will openly provide their software, because they want to protect their intellectual property. Although the MeDL has been provided to all project partners, the TaDLs have not been provided. The *system integrator* has no information about the task execution at the nodes, not even information about the *time budget* resp. the WCET of the tasks. Therefore testing is the only method, that enables the *system integrator* to validate the response requirements.

3.3.2.2 Synthesis of communication/component schedule

Response time requirements are usually given prior to system development. If the system is developed with today's tools, violations to severe latency restrictions are likely to happen. So the developers have to fine-tune the settings in order to find the right configuration for schedules that meet the response time requirements.

A better approach would be to specify the response time requirements directly in the model and to make the tools use this information to find the

appropriate settings to automatically generating schedules that meet the requirements.

To evaluate whether this is feasible, we now consider several relevant settings:

Basic Information Basic information, such as the general cluster layout, and fixed values, such as the transmission speed of the physical layer and the WCET of the tasks, must be provided by the developer. Most of this fundamental information cannot be generated automatically and in case of the WCET this would be very difficult as mentioned in Section 3.1.3 on page 74.

Message Period The message period influences the schedules so it affects the *information flow latency* (*IFL*) as well, but mainly it is responsible for the *event recognition latency* (*ERL*). As mentioned, the worst-case ERL is the period of the task that is connected to the corresponding sensor to detect the event resp. the change of state. This period is inherited from the message period of the message the sensing task will send, because the task should exactly produce as many messages as there are sent via the TTP bus by the TTP controller. Hence the worst-case ERL is the message period. This period is bounded by the `d_period` attribute. The decision for the sampling rate of the sensor, which causes the ERL, is fundamentally dependent of the system and hence cannot be made an automatic process.

TDMA round period and cluster cycle As a calculation the TDMA round period can be derived from all specified message periods:

$$\texttt{tr\_period} = GCD(\left\{\texttt{d\_period}_{mj} | mj \text{ is message } j\right\})$$

i.e., the *greatest common devisor* of all message periods. Therefore the cluster cycle period becomes

$$\texttt{cluster\_period} = LCM(\left\{\texttt{d\_period}_{mj} | mj \text{ is message } j\right\})$$

i.e., the *least common multiple* of all message periods.

This could lead to a bad proportion of the round and the cluster period. Consider an example of three different message flows with the message periods of 1.5 ms, 2 ms and 3 ms respectively. The TDMA round period would become 0.5 ms while the cluster period would be 6 ms. One cluster cycle would comprise 12 TDMA rounds, which is rather long, especially as only 9 messages could be transmitted and some rounds might transmit no sensible message at all.

Such cluster cycles would hardly be accepted by the developer as they get too complex, and empty rounds are not allowed anyway. Therefore the current workflow of the tools specifies the values the other way round: First specify the TDMA round period, then specify the message periods. The message periods will then be decreased automatically to fit a multiple of the TDMA round period. This way is sensible, as the developer has to decide the TDMA round period and therefore the resulting proportion to the cluster cycle by his or her own criteria. The system developer should decide, which message periods can be reduced in order to get a better proportion. In the example we could decrease the period of 1.5 ms to 1 ms and 3 ms to 2 ms resulting in a TDMA round period of 1 ms and a cluster cycle period of 2 ms, where one cluster cycle only comprises 2 TDMA rounds.

All the messages in one flow of information will have the same message period. Therefore the manual tuning of TDMA round period and of message period will not cost much effort, if there are little but complex flows of information. Hence automatic calculation will unlikely lead to the desired result and only save little effort and the impact to the IFL can hardly be considered.

PSD and PRD Setting the `max_PSD` and `max_PRD` attributes obviously leads to the desired effect of bounding the IFL. Consider the Formula 3.1 on page 80 that calculates the IFL. The WCET of the tasks and the durations of the message sending are fixed. Hence the only variable factors that influence the IFL are the durations, while the messages stay untouched in the CNI of the nodes. These values get fixed by a certain task schedule connected to a global communication schedule. Setting the optional attributes `max_PSD` and `max_PRD` will restrict the task schedulers to only produce schedules that under-run the specified values. This would ultimately lead to a specified IFL, and by setting the message period to the ERL the developer could force an end-to-end latency.

One question remains: how to distribute the PSD and PRD to the messages? We will call the sum of all PSD and PRD in an information flow the *idle message duration* (*IMD*):

$$IMD = \sum_{j} (PSD_{mj} + PRD_{mj})$$

Obviously the maximum IMD is given by the maximum IFL and the fixed values:

$$IMD_{max} = IFL_{max} - \left(\sum_{i} ti_{WCET} + \sum_{j} \Delta mj \right)$$

The developer could simply distribute the maximum allowed duration to all messages evenly. But only the sum of all PSD and PRD values affects the end-to-end latency, *i.e.*, the IMD. Hence explicit distribution of the IMD restricts the schedulers very much: all tasks have to be scheduled exactly that way that they meet their single PSD and PRD. This is a very severe restriction and will likely lead to unschedulable values, especially if the IMD is quite short. Evenly distributed IMD would force correct response times of the whole system, *if* the task schedulers could find a possible schedule. However, the minimum possible PSD and PRD value of a message is bounded by a specific MeDL: if a message $m1$ is consumed by a task t and a message $m2$ is produced, the PSD and PRD values are bounded by the fixed values of the MeDL and the task:

$$PRD_{m1} + PSD_{m2} \geq (m2_{send} - m1_{rec}) - t_{WCET}$$

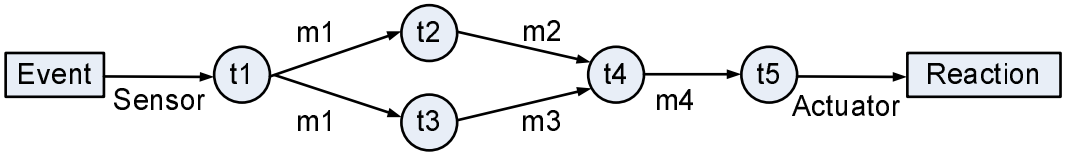
where $m1_{rec}$ is the point in time when $m1$ is received by the TTP controller of the node t and $m2_{send}$ is the point in time when the TTP controller starts to send $m2$. This means, the idle times of messages cannot be shorter than the period of time between one message is received and the next is sent decreased by the execution time of the task that processes the input message and produces the output message. Thus a given MeDL and evenly distributed IMD can lead to fundamentally unschedulable PRD and PSD values.

This is too severe for our purpose. As mentioned, the actual IMD must be below the maximum allowed IMD. However, it does not matter how the IMD is distributed to the different messages. It would be fully sufficient if one message gets a very short PSD and PRD while another message in the information flow gets bigger ones. This would give the schedulers the needed flexibility and if this leads to schedulable results, the developer would be content in respect to the response time requirements.

Unfortunately the dichotomy of TTP-Plan and TTP-Build thwarts this approach again: TTP-Plan builds the global communication schedule, the MeDL, without regarding timing requirements of the tasks. Single instances of TTP-Build use this MeDL to produce the task schedules, the TaDLs. Neither TTP-Plan nor the single instances of TTP-Build intercommunicate while building the schedules. The schedules are built fully independently only using the global restrictions given by TTP-Plan and the local restrictions given in each TTP-Build instance. Therefore a dynamic IMD distribution to the messages according to the needs of the tasks is fundamentally not possible. For rapid prototyping the scheduling algorithms must have been centralized with very great effort to allow dynamic IMD distribution. This centralization would either destroy the desired dichotomy of the two tools or introduce a very high intercommunication between the tools and the need of parallel scheduling.

Anyway this method will not be feasible if the dichotomy of *system integrator* and *subsystem supplier* must be kept in mind. The spatial and temporal separation of the scheduling processes make the dynamic IMD distribution impossible at all events.

Message depends on Message The relation `Message_depends_on_Message` is the only way to specify a flow of information in the current tools. Any arbitrary flow can be modelled using this relation. Consider the following message flow:



This would lead to the following relations:

- $m2$ depends on $m1$
- $m3$ depends on $m1$
- $m4$ depends on $m2$
- $m4$ depends on $m3$

Each of the relations has the additional attribute `generation_lag` which describes the next message generation at which the message must have been produced. This was mentioned before in Section 3.3.1 on page 78.

Assume that the `generation_lag` of the first relation is 0. The schedulers will schedule the task $t2$ that way that it finishes and writes its output until the next occurrence of $m2$ in relation to the input $m1$. If the `generation_lag` is 1, $t2$ may produce its output till the second occurrence of $m2$ in relation to $m1$ and so on. With increasing `generation_lag` the scheduler gains more flexibility. A `generation_lag` of 0 is a very severe restriction and will probably lead to unschedulable results.

The relation `Message_depends_on_Message` is a good way to model information flow in a TTA. But, again, it does not help in considerations about the end-to-end latency. And the reasons are the same as above: The information flow is considered only piecewise for each message. The `generation_lag` is set for each piece, not for the whole flow. Furthermore the `generation_lag` is no absolute value and depends on the message periods and the different phases of the messages in the TDMA round. The information restricts the scheduler

very much, too much, as the allowed lag is not freely distributed over all the messages in the message flow.

This is due to the same reasons stated above about the PSD and PRD: The dichotomy of global information and local information in the tools does not allow dynamic distribution of the idle time.

3.3.2.3 Possible Integration into the existing workflow

Synthesizing the system configuration from the response time requirements would inherently need a centralized scheduling, as both the global and the local properties influence the response time. This would destroy the wanted dichotomy of the scheduling process and therefore will unlikely find acceptance. Rapid prototyping with a modelling suite, such as SCADE or Simulink, uses both parts of the schedulers together. But they use the schedulers as they are, sequentially. Regarding the end-to-end latency in a centralized scheduler would be quite easy. The maximum IMD could be provided by the developer, resp. derived from the maximum IFL and the fix values. The scheduler would simply add the restriction that the sums of all PSD and PRD in a message flow may not exceed this IMD. In a *generate and test* methodology, which is used by *logic constraint solvers*, this restriction would automatically be regarded and the resulted schedules would meet the required response times.

Combining the schedulers only for rapid prototyping could mean a bad cost-effectiveness ratio for TTTech. As the schedulers are proprietary, they have to evaluate if adding intercommunication during the scheduling process could be introduced with justifiable effort.

Analysis of the system regarding the response time requirements could be added more easily, but is still not trivial. The global communication schedule as well as the local task schedules must be merged in order to look at a specific information flow. As *subsystem suppliers* want their intellectual property secured, they might not freely provide their software. Export scripts for TTP-Build could be used to export only the relevant information about the task scheduling. The *system integrator* would collect all this information from the suppliers and merge it for analysis.

Rapid prototyping with a modelling suite would ease this process, because suppliers and integrator are one instance. This feature could be added to Matlink for Simulink and TTPlink for SCADE respectively. The tool could automatically import all schedules and check the response times.

Additionally a better way of specifying a flow of information should be provided by the modelling suites. A new graphical notation as used in this section or even textual descriptions could be employed.

Maybe the graphical representation of the modelling suite could be enriched

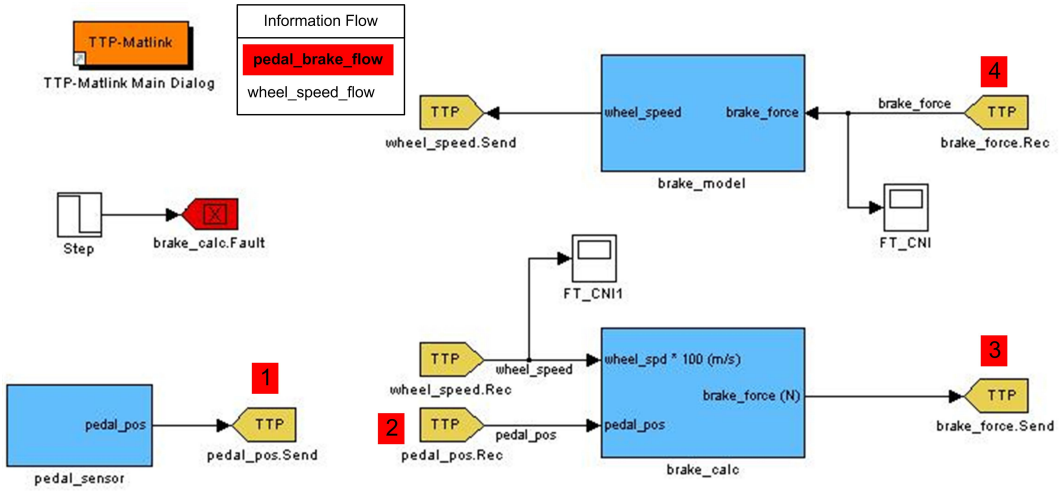


Figure 3.7: Suggested visualization and/or specification of information flow with Matlink

by adding information flow identifiers as shown in figure 3.7. A selected identifier would allow to specify an information flow by adding numbers to messages that indicate the flow relations.

At least the `Message_depends_on_Message` relation should be made accessible and augmented by some variable that indicates the maximum latency within the whole message flow.

Any of these methods would give the developer the opportunity to quickly and intuitively specify the flow of information. An analysis function could then combine all generated schedules and analyse the resulted latencies.

Best solution would be a toolchain that comprises centralized scheduling for MeDL and TaDLs. That would enable the tools to automatically consider dynamic distribution of *idle message durations* or the `generation_lag` by an automatic *generate and test* methodology.

However, this method is connected to a very high modification of the current tools. The analysis function could be realized with much less effort and provide a helpful function. The developer could manually review the response times of his system after the schedules have been built. This would lead to a little larger *generate and test* development loop.

4 Conclusion

We have described the model-based system design of a time-triggered architecture and listed the process in some detail regarding the dichotomy of global specifications done by TTP-Plan, local specifications done by TTP-Build and their conflation in a modelling suite such as Matlab Simulink resp. SCADE of Esterel Technologies.

We are modelling a time-triggered architecture in the DECOS research project: a high-lift flap system to demonstrate the results of the DECOS development. We listed what we have done so far and what is planned for the future in this context.

The practical application of the modelling tools raised the question on response time requirements. How can modelling tools handle end-to-end latency and how do they do it nowadays? Do they do all that is possible? Is it not possible to handle it after all?

We tackled this question by analysing the origin of latency in real-time systems in general and in the TTA in particular, where we depicted the worst-case latency. We introduced the division of the end-to-end latency into event recognition latency and information flow latency.

We analysed the workflow of the current modelling tools with respect to the response time requirements.

There we mentioned the question about worst-case execution times and depicted how Matlink of TTTech tries to handle this. It offers a nice and easy to use feature to test the average execution times, but does no adequate formal analysis, that would be sufficient for safety-critical applications.

We discussed which properties are critical to the end-to-end latency in a TTA and how the settings of the modelling tools influence these properties.

We showed that both, validation of the system response requirements and synthesis of the system configuration from the response time requirements, are not possible with the current toolchain. The main reason is the dichotomy of the global cluster information and the local node information that manifests in the partition of TTP-Plan and TTP-Build.

We depicted that synthesis is inherently not possible without immense effort by centralizing the scheduling. Combining the schedulers only for rapid prototyping could mean a bad cost-effectiveness ratio for TTTech, because

the dichotomy of *system integrator* and *subsystem suppliers* is sensible in the automotive domain.

Additionally we introduced how the conflation of the two TTTech tools in a modelling suite could help to get a convenient response time analysis function after all and therefore the modelling process could be enriched and save a lot of testing efforts.

A Latency Overview

Responsibilities

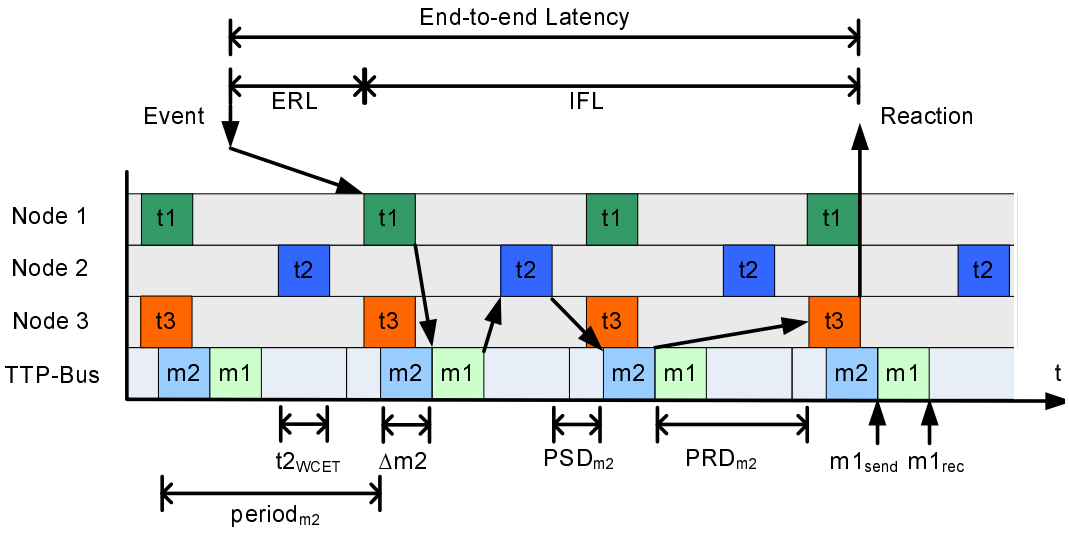
System Integrator

- Global system configuration:
  - amount of subsystems;
  - messages, mapping of messages to receiving and sending subsystems;
  - mapping of subsystems to nodes;
- global communication schedule, the MeDL:
  - physical transmission speed;
  - TDMA round length;
  - phases and periods of all messages;
- global timing constraints for single messages, *i.e.*, maximum values for PSD and PRD.

Subsystem Supplier Possibly different suppliers for different nodes!

- Local node configuration
  - choice of OS;
  - fragmentation of one subsystem to one or more tasks;
  - implementation of tasks, specification of WCETs;
- local task schedule, the TaDL;

→ actual PSD and PRD values get fixed.



Terms

End-to-End Latency The latency between the occurrence of an event resp. a change of state of the environment and the reaction of the system.

ERL – Event Recognition Latency The latency introduced by the execution period of the sensor task, *i.e.*, the period of time that passes until the event is recognized by a task.

IFL – Inter Flow Latency The (static) latency introduced by the flow of information in the system. It comprises the task execution times of the corresponding tasks, the message transfer duration of the corresponding messages and all PSD and PRD values of the messages.

PSD – Pre-Send Duration The duration that the message lies untouched in the CNI of a node after it has been written by a task and before it gets transmitted by the TTP controller.

PRD – Post-Receive Duration The duration that the message lies untouched in the CNI of a node after it has been received by a TTP controller and before it gets consumed by a task of the node.

IMD – Idle Message Duration Describes the period of time in a flow of information when the information lies untouched in the CNI of some node, *i.e.*, the period of time when the information does not get processed but is idle. It is the sum of the PSD and PRD values of all messages in the flow of information.

$\mathbf{t_{WCET}}$ The *time budget* of a task, *i.e.*, its worst-case execution time.

Δm – Message Transfer Duration The duration that is needed to transfer the message via the TTP bus. It comprises the reading of the value from the CNI, internal checks by the TTP controller, physical transfer, writing to the CNI of the receiver until it becomes available by the host application of the receiver. Usually more than one message is sent in one time slot by one node. Hence this value is the whole slot time, because all messages are processed all at once.

$\mathbf{m_{send}}$ The point in time when the TTP controller starts sending the message. Any task that produces this message must have finished the transfer to the CNI up to this point in time.

$\mathbf{m_{rec}}$ The point in time when the TTP controller has finished the receiving of a message and has stored the message in the CNI. This is the earliest point in time when an application task can start consuming the message.

$\mathbf{period_m}$ – Message Period The period of time at which the message regularly gets transmitted. The `d_period` is given by the developer and denotes the maximum allowed period and the `a_period` denotes the actual period that the system derived of the TDMA round size. It is `a_period` $\leq$ `d_period`.

B Acronyms

A/C	Aircraft
ACE	Actuator Control Electronics
AFDX	Avionic Full Duplex Switched Ethernet
CAU	Christian-Albrechts-Universität
CNI	Communication Network Interface
CSB	Cross Shaft Brake
COTS	Commercial-Of-The-Shelf
DECOS	Dependable Embedded Components and Systems
DLL	Dynamic Link Library
ECU	Electric Control Unit
ERL	Event Recognition Latency
FMEA	Failure Mode and Effects Analysis
FTCOM	Fault Tolerant Communication
FSM	Finite State Machine
IFL	Information Flow Latency
IMD	Idle Message Duration
KCG	Esterel Technologies' Qualified Code Generator
LIB	Static C Library
MCE	Motor Control Electronics
MeDL	Message Descriptor List
PPU	Position Pickoff Unit
PRD	Post-Receive Duration
PSD	Pre-Send Duration
RPM	Revolutions per Minute
RTW	Real-Time Workshop
SCU	System Control Unit
SSM	Safe State Machine
TaDL	Task Descriptor List
TDMA	Time Division Multiple Access
TTA	Time-Triggered Architecture
TTP	Time-Triggered Protocol
TTP/C	Time-Triggered Protocol version C
TUHH	Technische Universität Hamburg-Harburg
WCET	Worst-Case Execution Time
WCL	Worst-Case Latency

C Version Information

We used the following versions of tools at this work:

Distributor	Product	Version	
Mathworks	MATLAB	7.0.1	(R14SP1)
Mathworks	Simulink	6.1	(R14SP1)
Mathworks	Real-Time Workshop	6.1	(R14SP1)
Mathworks	RTW Embedded Coder	4.1	(R14SP1)
Mathworks	Stateflow	6.1	(R14SP1)
TTTech	TTP-Matlink	2.2	(R7.1)
TTTech	TTP-Calibrate	2.2	(R7.1)
TTTech	TTP-Plan	5.2	(R7.1)
TTTech	TTP-Build	5.2	(R7.1)
TTTech	TTP-Load	6.2	(R7.1)
TTTech	TTP-View	6.2	(R7.1)
Esterel Technologies	SCADE Suite	5.0 beta2	(build i24)

Basis of the model are the test bench requirements version 1.0r [Koc04].

Bibliography

[And96a] Charles André. Representation and Analysis of Reactive Behaviors: A Synchronous Approach. In *Computational Engineering in Systems Applications (CESA)*, pages 19–29, Lille (F), July 1996. IEEE-SMC. Available from World Wide Web: `http://www.i3s.unice.fr/~andre/CAPublis/Cesa96/SyncCharts_Cesa96.pdf`.

[And96b] Charles André. SyncCharts: A Visual Representation of Reactive Behaviors. Technical Report RR 95–52, rev. RR (96–56), I3S, Sophia-Antipolis, France, Rev. April 1996. Available from World Wide Web: `http://www.i3s.unice.fr/~andre/CAPublis/SYNCCHARTS/SyncCharts.pdf`.

[And03] Charles André. Semantics of S.S.M (Safe State Machine). Technical report, I3S, Sophia-Antipolis, France, 2003. Available from World Wide Web: `http://www.esterel-technologies.com/v3/?id=50399\&dwnID=48`.

[ARI] ARINC, Annapolis, Maryland, USA. *ARINC 664, Aircraft Data Networks, Part 7 — Deterministic Networks*. Available from World Wide Web: `http://www.arinc.com`.

[BA81] J. Dean Brock and William B. Ackerman. Scenarios: a model of non-deterministic computation. *Lecture Notes in Computer Science*, Formalization of Programming Concepts(107):252–259, 1981.

[BdS91] Frederic Boussinot and Robert de Simone. The ESTEREL language. another look at real time programming. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

[Ber00] Gerard Berry. The foundations of Esterel. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, 2000. Editors: G. Plotkin, C. Stirling and M. Tofte.

[BMH+01] Ralf Belschner, Robert Mores, Gary Hay, Josef Berwanger, Christian Ebner, Sven Fluhrer, Emmerich Fuchs, Bernd Hedenetz, Andreas Krüger, Peter Lohrmann, Dietmar Millinger, Martin Peller, Jens Ruh, Anton Schedl, and Michael Sprachmann. FlexRay: The communication system for advanced automotive control systems. In *SAE 2001 World Congress*, Detroit, USA, March 2001. Society of automotive Engineers.

[BPG00] Josef Berwanger, Martin Peller, and Robert Griessbach. byteflight - A new protocol for safety critical applications. In *Proc. FISITA 2000 Fédération Internationale des Sociétés d'Ingénieurs des Techniques de l'Automobile*, June 2000.

[BWH+03] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Paserone, and Alberto L. Sangiovanni-Vincentelli. Metropolis: an integrated electronic system design environment. *IEEE Computer*, 36(4):45–52, April 2003.

[DEC] DECOS - Dependable Components and Systems. Research project homepage. `https://www.decos.at/`.

[EES01] Jakob Engblom, Andreas Ermedahl, and Friedhelm Stappert. A worst-case execution-time analysis tool prototype for embedded real-time systems. In *Workshop on Real-Time Tools (RTTOOLS'2001), affiliated to CONCUR'2001*, Aalborg, Denmark, 2001.

[Est] Esterel Technologies. Company homepage. `http://www.esterel-technologies.com`.

[FMD+00] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, and Michael Walther. Time triggered communication on CAN. `http://www.can-cia.org/can/ttcan/fuehrer.pdf`, 2000.

[Gei04] Hendrik Geilsdorf. DECOS - TUHH - Regelungsstrategie. Internal DECOS Report, October 2004.

[GJ92] Robert Gabriel and Stefan Jähnichen. Graphical representations and software engineering. In *Proceedings of the Third Workshop on Future Trends of Distributed Computing Systems*, pages 234–239. IEEE Computer Society Press, April 1992.

[Han05] Reinhard von Hanxleden. Modellierung reaktiver systeme - statecharts und synchrone sprachen. In Peter Liggesmeyer and Dieter Rombach, editors, *Software Engineering für Eingebettete Systeme*. Spektrum Akademischer Verlag, 2005.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HCRP91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991. Available from World Wide Web: `citeseer.nj.nec.com/halbwachs91synchronous.html`.

[HdR91] Cornelis Huizing and Willem-Paul de Roever. Introduction to design choices in the semantics of statecharts. *Inf. Process. Lett.*, 37(4):205–213, 1991.

[Hex03] René Hexel. Fits: a fault injection architecture for time-triggered systems. In *CRIPTS '16: Proceedings of the twenty-sixth Australasian computer science conference on Conference in research and practice in information technology*, pages 333–338. Australian Computer Society, Inc., 2003.

[HG92] Cornelis Huizing and Rob Gerth. Semantics of reactive systems in abstract time. In *Proceedings of the Real-Time: Theory in Practice, REX Workshop*, pages 291–314. Springer-Verlag, 1992.

[HP85] David Harel and Amir Pnueli. On the development of reactive systems. *Logics and models of concurrent systems*, pages 477–498, 1985.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, 1979.

[ISO93] International Standards Organisation. *ISO 11898. Road Vehicles—Interchange of digital information—Controller area network (CAN) for high speed communication*, 1993.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[KB03] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[KG92] Hermann Kopetz and G. Grünsteidl. TTP - a time-triggered protocol for fault-tolerant real-time systems. Technical report, Institut für Technische Informatik, Technische Universität Wien, Treilstr. 3/182/1, A-1040 Vienna, Austria, 1992.

[KLMP82] Hermann Kopetz, Frieder Lohnert, Wolfgang Merker, and Georg Pauthner. The architecture of MARS. Research Report 1/1982, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1982.

[KN95] Hermann Kopetz and Roman Nossal. The cluster compiler - a tool for the design of time-triggered real-time systems. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 108–116, 1995.

[Koc04] Jens Koch. *Requirements Specification for the SP6 Test-bench (WP 6.1)*. Airbus Deutschland GmbH, DECOS Project, 1.0r edition, December 2004.

[Kop97] Hermann Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Kop03] Hermann Kopetz. The Rational of the DECOS Project. Internal DECOS report, Institut für Technische Informatik, Technische Universität Wien, Treilstr. 3/182/1, A-1040 Vienna, Austria, May 2003.

[Kop04] Hermann Kopetz. An integrated architecture for dependable embedded systems. In *SRDS*, pages 160–161, 2004.

[Lee03] Edward Ashcroft Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003. Available from World Wide Web: `http://ptolemy.eecs.berkeley.edu/publications/papers/03/overview`.

[LIN00] LIN Consortium. *LIN Protocol Specification 1.1*, April 2000. Available from World Wide Web: `http://www.lin-subbus.org`.

[LM87] Edward Ashford Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, September 1987.

[LP95] Edward Ashford Lee and Thomas M. Parks. Dataflow process networks. In *Proceedings of the IEEE*, pages 773–799, May 1995.

[Mat93] Mathworks Inc. *MATLAB reference guide*. Natick MA., 1993.

[Mat04] Mathworks Inc. *Stateflow and Stateflow Coder for use with Simulink — User's Guide*, 6 edition, 2004.

[Mer01] Agathe Merceron. Proving "no cliques" in a protocol. In *ACSC '01: Proceedings of the 24th Australasian conference on Computer science*, pages 134–139. IEEE Computer Society, 2001.

[MML97] Sharad Malik, Margaret Martonosi, and Yau-Tsun Steven Li. Static timing analysis of embedded software. In *DAC '97: Proceedings of the 34th annual conference on Design automation*, pages 147–152. ACM Press, 1997.

[MMP98] Agathe Merceron, Monika Müllerburg, and G. Michele Pinna. Verifying a time-triggered protocol in a multi-language environment. *Lecture Notes in Computer Science*, 1516:185–195, 1998.

[MMP99] Agathe Merceron, Monika Müllerburg, and G. Michele Pinna. "no collision" in a protocol with n stations: a comparative study of formal proofs. In S. Gnesi and D. Latella, editors, *Fourth Intern. ERCIM Workshop on Formal Methods for Industrial Critical Systems*, pages 153–177, Pisa, 1999.

[Neu02] Tom Neuheuser. Modellbildung, -validierung und Entwurf des Regelungskonzeptes eines Antriebssystems auf Basis elektrischer Stellglieder für ein einzelnes Landeklappensegment eines Transportflugzeuges. Diplomarbeit (Flugzeug-Systemtechnik, Technische Informatik), Technische Universität Hamburg-Harburg, February 2002.

[NL94] Roman Nossal and L. Lechner. Simulation of a time-triggered communication protocol for fault-tolerant real-time systems. Technical report, Institut für Informatik, Technische Universität Wien, Vienna, Austria, August 1994.

[NL02] Roman Nossal and Roland Lang. Model-based system development - an approach to building X-by-wire applications. *IEEE Micro*, 22(04):56–63, July/August 2002.

[OSE01] OSEK. *OSEK/VDX Time-Triggered Operating System Specification 1.0*, 1.0 edition, July 2001.

[PH04] Steffen Prochnow and Reinhard von Hanxleden. Visualisierung komplexer reaktiver Systeme – Annotierte Bibliographie. Technischer Bericht 0406, Christian-Albrechts-Universität Kiel, Institut für Informatik und Praktische Mathematik, June 2004.

[PSvH99] Holger Pfeifer, Detlef Schwier, and Friedrich W. von Henke. Formal verification for time-triggered clock synchronization. In *DCCA '99: Proceedings of the conference on Dependable Computing for Critical Applications*, page 207. IEEE Computer Society, 1999.

[RIS] RISE - Reliable Innovative Software for Embedded Systems. Research project homepage. `http://www.esterel-technologies.com/rise/`.

[RTC92] RTCA/EUROCAE. *Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[Rus01] John Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, September 2001. Available from World Wide Web: `http://www.csl.sri.com/~rushby/abstracts/buscompare`.

[Ste] Stefan Jähnichen. Homepage: Technische Universität Berlin - Fachgebiet Softwaretechnik. `http://swt.cs.tu-berlin.de/`.

[TTA03] TTA-Group. *Time-Triggered Protocol TTP/C High-Level Specification Document Protocol Version 1.1*, 1.4.3 edition, November 2003.

[TTT] TTTech. Company homepage. `http://www.tttech.com`.

[Uni] University of California Berkeley — EECS - Electrical Engineering and Computer Sciences. University homepage. `http://www.eecs.berkeley.edu/`.