15

Development Tools

P. Pop

Technical University of Denmark

A. Goller

TTTech Computertechnik AG

T. Pop

Ericsson AB

P. Eles

Linköping University

CONTENTS

15.1 Introduction

Embedded systems are now everywhere: From medical devices to vehicles, from mobile phones to factory systems, almost all the devices we use today are controlled by embedded computers. Over 98% of microprocessors are used in embedded systems, and the number of embedded systems in use has become larger than the number of humans on the planet, and is projected to increase to 40 billion worldwide by 2020 [11, 84]. The embedded systems market size is about 100 times larger than the desktop market, with over 160 billion Euros worldwide and a growth rate of 9% [84].

The complexity of embedded systems is growing at a very high pace and their constraints in terms of performance, reliability, cost and time-to-market are getting tighter. The embedded software size is increasing 10 to 20% per year, depending on the application area. Today's cars have more than 100 million object code instructions [84], while in avionics, the size of the certified software has increased from 12 Mbytes in Airbus A340 to 80 Mbytes in A380 [11].

At the same time, high complexity, increasing density and higher operational frequencies have led to an increasing number of faults [65]. Embedded systems are increasingly used in safety-critical contexts, such as automotive applications, avionics, medical equipment, control and telecommunication devices, where any deviation from the specified functionality can have catastrophic consequences. In addition, many industries are very cost-sensitive, and thus the dependability requirements have to be met within a tight cost constraint.

Therefore, the task of designing such systems is becoming increasingly important and difficult at the same time. The difficulty of designing embedded systems is reflected by the share of the development and implementation costs from the final product price, which is 36% in the automotive area, 22% in industrial automation, 37% in the telecommunications area, 41% in consumer electronics and 33% for medical equipment [276]. This has led to a *design productivity gap*: The number of on-chip transistors is growing each year by 58% (according to Moore's law), whereas the productivity of hardware designers is only growing by 21% per year, and the software productivity is lagging even further behind [276].

Many organizations, including automotive manufacturers, are used to designing and developing their systems following some version of the "waterfall" [94] model of system development. This means that the design process starts with a specification and, based on this, several system-level design tasks are performed manually, usually in an ad-hoc fashion. Then, the hardware and software parts are developed independently, often by different teams located far away from each other. Software code is written, the hardware is synthesized and they are supposed to integrate correctly. Simulation and testing are done separately on hardware and software, respectively, with very few integration tests.

If this design approach was appropriate when used for relatively small systems produced in a well-defined production chain, it performs poorly for more complex systems, leading to an increase in the time-to-market. New approaches and tools have been proposed, which are able to: Successfully manage the complexity of embedded systems, meet the constraints imposed by the application domain, shorten the time-to-market, and reduce development and manufacturing costs. There are many development tools, and their use depends on the application area. The most important embedded systems tools are presented in [191].

In the next section, we present the typical design tasks, emphasizing the communication synthesis task, which is the focus of this chapter. We will present state-of-the-art techniques and tools for the communication scheduling and communication configuration. In Section 15.3, we will define the general problem of scheduling, discuss its complexity and the typical strategies employed. Once a schedule is generated, it can be manipulated, extended and visualized.

As we will show, communication synthesis has a strong impact at the system-level. In this context, in Section 15.4, we will discuss the integrated (holistic) scheduling of tasks and messages, and the bus schedule optimization to support the fulfillment of timing constraints. Systems are seldom built from scratch, hence, in Section 15.5 we discuss the issues related to incremental design, where a schedule has to be generated such that it is flexible, i.e., supports the addition of new functionality. Although this book is focused on time-triggered systems, using an event-triggered approach at the processor level can be the right solution under certain circumstances [205]. Hence, in Section 15.6, we present an approach to integrate event-driven tasks with a time-triggered communication infrastructure.

Once a schedule is generated, it has to be translated into a communication configuration, particular for the communication protocol used, such as TTP[1] or FlexRay. In Section 15.7 we illustrate this issue using the tool chain from TTTech. Finally, in the last section of this chapter, we discuss verification and certification aspects.

15.2 Design Tasks

The aim of a design methodology is to coordinate the design tasks such that the time-to-market is minimized, the design constraints are satisfied and various parameters are optimized. The following are the state-of-the-art methodologies in embedded systems design:

- **Function/architecture co-design:** Function/architecture co-design is a design methodology [162, 323] which addresses the design process at higher abstraction levels. Function/architecture co-design uses a top-down synthesis approach, where trade-offs are evaluated at a high level of abstraction. The main characteristic of this methodology is the use, at the same time with the top-down synthesis, of a bottom-up evaluation of design alternatives, without the need to perform a full synthesis of the design. The approach to obtain accurate evaluations is to use an accurate modeling of the behavior and architecture, and to develop analysis techniques that are able to derive estimates and to formally verify properties relative to a certain design alternative. The determined estimates and properties, together with user-specified constraints, are then used to drive the synthesis process.

  Thus, several architectures are evaluated to determine if they are suited for the specified system functionality. There are two extremes in the degrees of freedom available for choosing an architecture. At one end, the architecture is already given, and no modifications are possible. At the other end of the spectrum, no constraints are imposed on the architecture selection, and the synthesis task has to determine, from scratch, the best architecture for the required functionality. These two situations are, however, not common in practice. Often, a *hardware platform* is available, which can be *parameterized* (e.g., size of memory, speed of the buses, etc.). In this case, the synthesis task is to derive the parameters of the architecture such that the functionality of the system is successfully implemented. Once an architecture is determined and/or pa-

[1] Throughout this chapter, we use “TTP” instead of “TTP/C,” as it is the commercial and more customary term.

rameterized, the function/architecture co-design continues with the mapping of functionality onto the instantiated architecture.

- **Platform-based design:** In order to reduce costs, especially in the case of a mass market product, the system architecture is usually reused, with some modifications, for several product lines. Such a common architecture is denoted by the term *platform*, and consequently the design tasks related to such an approach are grouped under the term platform-based design [163].

  One of the most important components of any system design methodology is the definition of a *system platform*. Such a platform consists of a hardware infrastructure together with software components that will be used for several product versions, and will be shared with other product lines, in the hope to reduce costs and the time-to-market.

  The authors in [163] have proposed techniques for deriving such a platform for a given family of applications. Their approach can be used within any design methodology for determining a system platform that later on can be parameterized and instantiated to a desired system architecture.

  Considering a given application or family of applications, the system platform has to be instantiated, deciding on certain parameters, and lower level details, in order to suit the particular application(s). The search for an architecture instance starts from a certain platform, and a given application. The application is mapped and compiled on an architecture instance, and the performance numbers are derived, typically using simulation. If the designer is not satisfied with the performance of the instantiated architecture, the process is repeated.

- **Incremental design process:** A characteristic of the majority of approaches to the design of embedded systems is that they concentrate on the design, from scratch, of a new system optimized for a particular application. For many application areas, however, such a situation is extremely uncommon and appears only rarely in design practice. It is much more likely that one has to start from an already existing system running a certain application and the design problem is to implement new functionality (including also upgrades to the existing one) on this system. In such a context, it is very important to operate no, or as few as possible, modifications to the already running application. The main reason for this is to avoid unnecessarily large design and testing times. Performing modifications on the (potentially large) existing application increases design time and, even more, testing time (instead of only testing the newly implemented functionality, the old application, or at least a part of it, has also to be retested) [264].

  However, minimizing the modification cost is not the only aspect to be considered. Such an incremental design process, in which a design is periodically upgraded with new features, is going through several iterations. Therefore, after new functionality has been introduced, the resulting system has to be implemented such that additional functionality, later to be mapped, can easily be accommodated [264].

There is a large body of literature on systems engineering that discusses various methodologies for systems development. Many methodologies employed in the development of safety-critical systems are a variant of the “V-Model” [94], named after the graphical representation in a “V” shape of the main development phases, that starts with the requirements phase, followed by hazard and risk analysis, specification, architectural design, module design, module construction and testing (at the bottom of the “V” shape), system integration and testing, system verification, system validation and, finally, certification. For example, the V-model is employed in the SETTA approach [6], which proposes system development methodologies for time-triggered systems in the automotive and aerospace domains.

The design tasks that have to be performed depend on the type of system being developed and on the design methodology employed. For safety-critical systems, the design tasks are often dictated by certification requirements, or by the development approach used. For example, the Automotive Open System Architecture (AUTOSAR) defines, besides the models for system development, the design tasks that have to be performed [18]. Regardless of the design tasks performed, *model-based design* is used throughout the development process: The interaction among design tasks is facilitated by the use of models, and the modeling is supported by graphical modeling tools. The following are the typical design tasks:

- **Functional analysis and design:** The functionality of the host system, into which the electronic system is embedded, is normally described using a formalism from that particular domain of application. For example, if the host system is a vehicle, then its functionality is described in terms of control algorithms using differential equations, which are modeling the behavior of the vehicle and its environment. At the level of the embedded real-time system which controls the host system, the functionality is typically described as a set of functions, accepting certain inputs and producing some output values.

  During the functional analysis and design stage, the desired functionality is specified, analyzed and decomposed into sub-functions based on the experience of the designer.

- **Architecture selection:** The architecture selection task decides what components to include in the hardware architecture and how these components are connected. Architecture selection relies heavily on the experience of the designer and previous product versions. If needed, new hardware components may be designed and synthesized, part of the **hardware design** task.

- **Mapping:** The mapping task has to decide what part of the functionality should be implemented on which of the selected components.

  The automotive companies integrate components from suppliers, and thus the mapping choices are often limited.

- **Software design and implementation:** This is the phase in which the software is designed and the code is written. The code for the functions is developed manually or generated automatically. The low-level software that interacts closely with the hardware is sometimes called *firmware*, and the task of designing it is hence called **firmware design**.

At this stage, the correctness of the software is analyzed through simulations, but no analysis of timing constraints is performed, which is done during the scheduling and schedulability analysis stage.

- **Scheduling and schedulability analysis:** Once the functions have been defined and the code has been written, the scheduling task is responsible for determining the execution order of the functions inside an ECU, and the transmission of messages such that the timing constraints are satisfied.

  Schedulability analysis is used to determine if an application is schedulable. A detailed discussion about scheduling and schedulability analysis is presented in the next section.

- **Integration:** In this phase, the manufacturer has to integrate the ECUs from different suppliers. The performance of the interacting functionality is analyzed using analysis tools and time-consuming simulation runs using the realistic environment of a prototype car.

  Detecting potential problems at such a late stage may lead to large delays in the time-to-market, since once a problem is identified, it takes a very long time to go through all the previous stages in order to fix it.

- **Communication synthesis:** Many real-time applications, following physical, modularity or safety constraints, are implemented using *distributed architectures*. The systems addressed in this book are composed of several different types of hardware components, interconnected in a network.

  In this context, an important design task is the communication synthesis task, which decides the scheduling of communications and the configuration parameters specific to the employed protocol. These decisions have a strong impact on the overall system properties such as predictability, performance, dependability, cost, maintainability, etc.

- **Calibration, testing, verification:** These are the final stages of the design process. If not enough analysis, testing and verification has been done in earlier stages of the design, these stages can be very time consuming, and problems identified here may lead to large delays.

15.3 Schedule Generation

According to [49], a *scheduling policy* provides two features: (i) an algorithm for ordering the use of system resources (in particular the processors, the buses, but also I/Os) and (ii) a means of predicting the worst-case behavior of the system when the scheduling algorithm is applied. The prediction, also known as *schedulability analysis*, can then be used to guarantee the temporal requirements of the application.

The aim of a schedulability analysis is to determine *sufficient* and *necessary* conditions under which an application is schedulable. An application is schedulable if there exists at least one scheduling algorithm that is able to produce a feasible schedule. A *schedule* is a particular assignment of activities to the resource (e.g., tasks to processors). A schedule is *feasible* if all tasks can be completed within the specified constraints. Before such techniques can be used, the worst-case execution times of tasks have to be determined. Tools such as aiT [98] can be used in order to determine the worst-case execution time of a piece of code on a given processor.

The analysis and optimization techniques employed depend on the scheduling policy and the model of the functionality used. The design techniques typically take as input a model of the functionality consisting of sets of interacting tasks. A *task* is a sequence of computations (corresponding to several building blocks in a programming language) which starts when all its inputs are available. When it finishes executing, the task produces its output values. Tasks can be *preemptible* or *non-preemptible*. Non-preemptible tasks are tasks that cannot be interrupted during their execution. Preemptible tasks can be interrupted during their execution. For example, a higher priority task has to be activated to service an event; in this case, the lower priority process will be temporarily preempted until the higher priority process finishes its execution. Tasks send and receive messages. Depending on the communication protocol, message transmission can be preemptible or non-preemptible. Large non-preemptible messages can be split into packets before transmission.

There are several approaches to scheduling:

- Non-preemptive *static cyclic scheduling* (SCS) algorithms are used to build, offline, a schedule table with activation times for each task (and message), such that the timing constraints of tasks (and messages) are satisfied.

- Preemptive *fixed priority scheduling* (FPS). In this scheduling approach, each task (and message) has a fixed (static) priority which is computed offline. The decision on which ready task to activate (and message to send) is taken online according to their priority.

- *Earliest deadline first* (EDF). In this case, that task will be activated (and that message will be sent) which has the nearest deadline.

For static cyclic scheduling, if building the schedule table fulfills the timing constraints, the application is schedulable. In the context of online scheduling methods, there are basically two approaches to the schedulability analysis: Utilization-based tests and response-time analysis.

- The *utilization tests* use the *utilization* of a task or message (its worst-case execution time relative to its period) in order to determine if the task sets (or messages) are schedulable.

- A *response time analysis* has two steps. In the first step, the analysis derives the worst-case response time of each task and message (the time it takes from the moment it is ready for execution, until it has finished executing). The second step compares the worst-case response time of each task and message to its deadline and, if the response times are smaller than or equal to the deadlines, the application is schedulable.

As mentioned throughout this book, another important distinction is between two basic design approaches for real-time systems, the event-triggered and time-triggered approaches.

- **Time-Triggered:** In the time-triggered approach, activities are initiated at predetermined points in time. In a distributed time-triggered system, it is assumed that the clocks of all nodes are synchronized to provide a global notion of time. Time-triggered systems are typically implemented using *non-preemptive static cyclic scheduling*, where the task activation or message communication is done based on a schedule table built offline.

- **Event-Triggered:** In the event-triggered approach, activities happen when a significant change of state occurs. Event-triggered systems are typically implemented using *preemptive priority-based scheduling*, or *earliest deadline first*, where, as response to an event, the appropriate task is invoked to service it.

In this chapter, we are interested in time-triggered systems implemented using non-preemptive static cyclic scheduling. A static schedule is a list of activities that is repeated periodically. Each activity has an associated start time, capturing, for example, when the particular task has to be activated or the message has to be transmitted. There are several types of schedules in time-triggered systems.

- **Message schedules:** These are the schedules for the messages and frames transmitted on the bus. The message schedules are organized according to a TDMA policy: Each processor can transmit only during a predetermined time interval, the so-called TDMA slot. In such a slot, a node can send several messages packaged in a frame (TTP), or even several frames (TTEthernet). Some protocols require a fixed sequence of slots, each slot corresponding to a node, and covering all the nodes in the architecture. This sequence is called a TDMA round. Several TDMA rounds can be combined together in a cycle that is repeated periodically (cluster cycle). Other protocols (like TTEthernet) are less strict and allow a basically arbitrary pattern within a cluster cycle. However, the design of control algorithms often implies the use of TDMA rounds, and several TDMA rounds with different length may be folded into a cluster cycle. The sequence and length of slots may be required to be the same for all TDMA rounds (FlexRay). In TTP, different lengths of slots are allowed, but a fixed sequence must be maintained.

- **Task schedules:** These are the schedules for tasks running on the processors, according to a SCS policy. Such a scheduling scheme is also called “timeline scheduling,” and is the most used approach to handle periodic tasks in safety-critical systems. The advantages and disadvantages of timeline scheduling (especially compared to fixed-priority preemptive scheduling) are well understood [203]. The tasks are repeated periodically, with a period called the *major cycle*. In most cases, the task periods are not identical, so the major cycle is set to the least common multiple of all periods, and is subdivided into *minor cycles*. A task with a smaller period will appear in several minor cycles, thus achieving its desired rate. The task schedules are implemented using a *cyclic executive*, typically based on a clock tick (an interrupt), which triggers the start of the minor cycle. Often, other interrupts are disabled (or severely limited) and when the tasks in the minor cycle finish executing, control is passed to a background scheduler that attends to less important activities.

- **Partition schedules:** In safety-critical systems, applications of different criticality levels are often separated from each other using spatial and temporal partitioning. Thus, with temporal partitioning, each application is allowed to run only within predefined time slots, allocated on each processor. The sequences of time slots for all applications on a processor are grouped within a *major frame*, which is repeated periodically.

- **Interrupt schedules:** While task and partition schedules mainly focus on the user application, interrupt schedules are used for middleware tasks. Certain actions, like reading and unpacking a frame, have to be executed actually for every frame received. An interrupt (or middleware task activation) therefore may occur several times within a cluster cycle or even within a TDMA round. The interrupt schedule specifies what specific actions to execute in this particular instance of an interrupt occurrence.

- **Cluster schedules:** To implement a schedule in a distributed system, a global notion of time is required. The previously mentioned schedules are typically specified at the cluster level, since clock synchronization is performed at the cluster level. A cluster schedule captures task, message and partition schedules within a cluster. Several cluster schedules can be present in a system, but they will not be synchronized with each other.

15.3.1 Requirements and Application Model

The requirements imposed on an embedded system depend on the particular application that it implements. Requirements are divided into functional requirements and non-functional requirements. The difficulty of designing embedded systems lies in the many competing non-functional requirements that have to be satisfied. Typical non-functional requirements are: Performance (in terms of latency, throughput, speedup), unit cost (the cost of manufacturing each copy of the system), non-recurring engineering cost (the one-time monetary cost of designing the system), size, power consumption, flexibility (how easy is it to change the functionality, to add new functions), time-to-prototype, time-to-market and dependability attributes such as reliability, maintainability and safety.

In a *real-time system*, the timing constraints are of utmost importance: “The correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced” [169]. In *hard* real-time systems, missing a deadline can lead to a catastrophic failure. Design methodologies for these systems are based on their worst-case execution times. In *soft* real-time systems, missing a deadline does not cause catastrophic failures in the system but leads to a certain performance degradation. The following are typical constraints imposed in a hard real-time system:

- **Timing constraints**. The worst-case execution time (WCET) C_i is an upper bound on the execution times of a task τ_i, which depends on its functionality and the particular processor N_i where it runs. Tasks can have constraints on their completion or activation. Thus, a *deadline* D_i of a task τ_i is a time at which the task must complete its execution. Tasks which must be executed once every T_i units of time are called periodic tasks, and T_i is called their period. (Each execution of a periodic task is called a job.) All other tasks are called aperiodic. *Release times* restrict the start time of task activations (often to avoid resource contention). Another important timing constraint, especially in the context of control applications, is *jitter*, which captures the time-variation of a periodic event. Note that all these constraints also apply to messages.

- **Precedence constraints:** They impose an ordering in the execution of activities. The behavior of the system is often modeled as a sequence of activities. Thus, before a task can start, it has to wait for the input from another task. For example, to perform an image recognition, first the image has to be acquired. *Distance* constraints express a minimum distance between two activities, on top of a precedence constraint. The opposite of distance constraints are the *freshness* constraints, which express the maximum distance between two consecutive activities. Freshness constraints are typically placed on sensor data.

- **Resource constraints:** To perform their function, tasks have to use resources. A task may have a *locality* constraint which requires the allocation of a task to a specific processor, for example, because it has to use an actuator attached to this particular processor. When several tasks want to use the same resource (e.g., shared memory), we impose *mutual exclusion* constraints. Messages exchanged between tasks on different processors have to use the bus, thus imposing *communication* constraints.

- **Extendability constraints:** Of specific interest are changes that are considered “local.” Such a local change is a new message m_{i+1} that shall be transmitted from one node A to another node B, but not to all other nodes C to Z. Ideally, the communication configuration of nodes C to Z need not be updated due to this change. A slightly different case is if message m_i, which only is transmitted between nodes A and B, gets changed in its size.

  Unfortunately, this view does not provide enough detail to decide whether this change is local or not. If it is necessary to move another message m_j due to the now bigger size of message m_i, it is obviously not simply a local change. Constraints may exist regarding the placement and alignment of messages within frames. A certain amount of bandwidth (per host) could be reserved for future extensions. Users may want to specify the layout of the frame manually, but leave the scheduling of the frames to a tool. The objective is to be able to modify and extend an existing schedule throughout the whole development and product lifetime just by local changes in order to save verification and certification efforts.

These requirements dictate the types of schedules that have to be produced, and the types of tools needed to generate the schedules. For example, the precedence constraints will capture if the interaction between components is synchronous or asynchronous. A fully synchronous application (the tasks and the communication are in phase and with the same speed) needs a more interacting design tool chain, that will produce synchronized cluster-level schedules for both tasks and messages, than an asynchronous application. There can be several setups, which will be reflected in the tools used and the tool flow employed: The time-triggered network communication and application are synchronous; the time-triggered network communication and application are asynchronous (causing oversampling and undersampling issues); the network communication is not time-triggered and the application is bound to a local clock (e.g., a control loop with CAN); and the network communication is not time-triggered and the application reacts on events.

Thus, in this section we discuss the tools needed for generating message schedules for time-triggered communication. In Section 15.4, we consider a complex setup, where tasks can be both time-triggered and event-triggered, and messages are transmitted using FlexRay, which has both static (time-triggered) and dynamic (event-triggered) segments. The assumption is that tasks and messages are synchronous. We discuss holistic scheduling: How to generate the cluster-level task and message schedules such that the timing constraints are satisfied for both time-triggered and event-triggered activities. We show how schedulability analysis has to be integrated with schedule generation to guarantee the timing constraints. In Section 15.5, we discuss how the schedules can be generated such that they are flexible, i.e., easy to extend with new functionality. Section 15.6 focuses on the interaction between event-triggered tasks, which produce event-triggered messages, and the time-triggered frames scheduled over TTP. Several approaches that schedule event-triggered messages over time-triggered frames are proposed and discussed. We propose both problem-specific heuristic algorithms and meta-heuristics for the optimization of the generated schedules. Section 15.3.2 discusses the complexity of the scheduling problem and the typical solutions employed. As we will show in the remainder of this chapter, the way the schedules are generated and optimized has a significant impact not only on the timing constraints, but also on flexibility, latency, jitter, buffer size, switching devices required and others.

15.3.1.1 Application Model

There is a lot of research in the area of system modeling and specification, and an impressive number of representations have been proposed. An overview, classification and comparison of different design representations and modeling approaches is given in [85]. The scheduling design task deals with sets of interacting tasks. Researchers have used, for example, dataflow process networks (also called task graphs, or process graphs) to describe interacting tasks, and have represented them using directed acyclic graphs, where a node is a process and the directed arcs are dependencies between processes.

In this subsection, we describe the application model assumed in the following sections. Thus, we model an application $\mathcal{A}$ as a set of directed, acyclic, polar graphs $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \mathcal{A}$. A node $\tau_{ij} \in \mathcal{V}_i$ represents the jth task or message in $\mathcal{G}_i$. An edge $e_{ijk} \in \mathcal{E}_i$ from τ_{ij} to τ_{ik} indicates that the output of τ_{ij} is the input of τ_{ik}. A task becomes ready after all its inputs have arrived, and it issues its outputs when it terminates. A message will become ready after its sender task has finished, and becomes available for the receiver task after its transmission has ended. The communication time between tasks mapped on the same processor is considered to be part of the task's worst-case execution time and is not modeled explicitly. Communication between tasks mapped on different processors is performed by message passing over the bus. Such message passing is modeled as a communication task inserted on the arc connecting the sender and the receiver task.

We consider that the scheduling policy for each task is known (either *SCS* or *FPS*), and we also know how the messages are transmitted. For example, for FlexRay, we would know if the message is sent in the static or dynamic segment. For a task $\tau_{ij} \in \mathcal{V}_i$, $Node_{\tau_{ij}}$ is the node to which τ_{ij} is assigned for execution. When executed on $Node_{\tau_{ij}}$, a task τ_{ij} has a known worst-case execution time $C_{\tau_{ij}}$. We also consider that the size of each message m is given, which can be directly converted into communication time C_m on the particular bus.

Tasks and messages activated based on events also have a priority, $priority_{\tau_{ij}}$. All tasks and messages belonging to a task graph G_i have the same period $T_{\tau ij} = T_{\mathcal{G}_i}$ which is the period of the task graph. A deadline $D_{\mathcal{G}_i}$ is imposed on each task graph $\mathcal{G}_i$. In addition, tasks can have associated individual release times and deadlines. If dependent tasks are of different periods, they are combined into a merged graph capturing all activations for the hyper-period (LCM of all periods) [261].

15.3.2 Scheduling Complexity and Scheduling Strategies

As mentioned earlier, a schedule defines the assignment of activities to the resources. The complexity of deriving a schedule depends on the type and quantity of resources available, the constraints imposed, and the objective function that has to be optimized. Scheduling is probably one of the most researched problems in computer science, and there is an enormous amount of results. There are several surveys available which present the scheduling problems, their complexity and the strategies used.

The following are the main findings regarding the complexity of the scheduling problems related to time-triggered systems, as reported in [300]:

- The integrated task and message scheduling problem to find the optimal schedule (the one with minimum length) is NP-complete. Thus, given a task graph model of the application, a limited number of processors interconnected by a time-triggered bus, the problem of finding a feasible schedule that minimizes the schedule length does not have a polynomial-time solution.

- The optimal task scheduling problem on a limited number of processors, but without considering the communication costs, is also NP-complete.

- The scheduling problem, considering communication costs, on an unlimited number of processors is NP-complete.

- The task scheduling problem, without the communication costs, is polynomial on an unlimited number of processors. Of course, there are never unlimited resources in a real system.

- The problem of deriving a schedule for messages, with the aim of optimizing a given design metric, is NP-complete if it can be reduced to the "knapsack" or "bin-packing" problems, which themselves are NP-complete.

These results mean that the schedules cannot be derived manually, and tool support is necessary. The scheduling problem is a very well-defined optimization problem, and has been tackled with every conceivable approach.

- **Mathematical techniques:** Researchers have proposed integer linear programming, mixed-integer programming and dynamic programming. Decomposition strategies (such as Benders-decomposition), enumerative techniques such as Branch-and-Bound and Lagrangian relaxation techniques have also been proposed. Such mathematical approaches have the advantage of producing the optimal solution. However, they are only feasible for limited problem sizes due to the prohibitive run times.

- **Artificial intelligence (AI):** AI techniques have been used for scheduling, such as expert/knowledge-based systems, distributed agents and neural networks.

- **Scheduling heuristics:** The most popular scheduling heuristics are *list scheduling* and *clustering* [300]. List scheduling (LS) is the dominant scheduling heuristic technique. LS heuristics use a sorted priority list, containing the tasks ready to be scheduled, while respecting the precedence constraints. A task is ready if all the predecessor tasks have finished executing and all the incoming messages are received. LS generates the schedule by successively scheduling each task (and message) onto the processor (bus). The start time in the schedule table is the earliest time when the resource is available to the respective task (or message). The allocation of tasks to processors has a direct influence on the communication cost. When the allocation of tasks to processors is not decided, *clustering* can be used to group tasks that interact heavily with each other, and allocate them on the same processor [300].

- **Neighborhood search:** Although very popular, the drawback of scheduling heuristics such as list scheduling is that they do not guarantee finding the optimal solution, i.e., they get stuck in a local optimum in the solution space. Neighborhood search techniques are meta-heuristics (i.e., they can be used for any optimization problem, not only scheduling) that can be used to escape from the local optimum. Neighborhood search techniques use design transformations (moves) applied to the current solution, to generate a set of neighboring solutions that can be further explored by the algorithm. Popular meta-heuristics in this category are Simulated Annealing, Tabu Search and Genetic Algorithms [46].

In the following subsections, we will use constructive heuristics such as list scheduling to generate schedules, and meta-heuristics (neighborhood search techniques) such as Simulated Annealing and Tabu Search to optimize a given schedule for a certain metric. In the next subsections, some concepts based on and extending the list-scheduling heuristic are discussed in detail. These concepts are partly implemented in the scheduler of $^{\mathrm{TTP}}$Plan [344], the cluster design tool for TTP clusters from TTTech. Lastly, we provide further details on the scheduling approach chosen for $^{\mathrm{TTP}}$Plan.

15.3.2.1 Incremental Scheduling

Once a schedule has been generated and optimized, an important aspect is the extension of a schedule. The goal is to keep the scheduled tasks or messages as they are, and to only add new tasks or messages in the free places. Incremental scheduling (a.k.a. *schedule extension*) thus means that scheduling is done in discrete steps.

Schedule Steps

Each time a schedule is made, this is called a “schedule step.” These schedule “steps” do not really form a sequence of *different* steps, but the whole process is a quite iterative one: After an initial schedule has been created, some properties or objects may be changed, and a new schedule is made, which is possibly analyzed. Due to this analysis or to change requests, further modifications are done, and a new schedule is made. Each such cycle of changing and scheduling is considered a *schedule step*. It is possible to make as many schedule steps as needed, until the result is satisfactory. The concept of schedule steps fits well into the list-scheduling approach as discussed above. Furthermore, a schedule step does not imply that already placed tasks or messages are kept in their places. Any modification of the output is possible.

Freezing and Thawing

One can keep a schedule by “freezing” the current schedule step. By adding new messages (with their type, period and further attributes, such as sender and receiver) to it, and scheduling again, the "holes" in the original schedule are filled without changing the already placed parts. The inverse operation is to "thaw" a schedule step. This means to actually throw away the schedule that was computed in this very step, but keeping the schedule parts from previous schedule steps. The additions made in this step are then merged with the new additions (made after the just thawed schedule step), and together considered the change set for the current schedule step. Obviously, only the last frozen schedule step can be thawed. The concept of freezing and thawing schedule steps also nicely fits into the list-scheduling approach.

Apart from adding new messages, other possible additions after a frozen schedule step are:

- Additional hosts and subsystems
- Additional message types
- Mapping of new subsystems to hosts

In $^{\text{TTP}}$Plan, only "frozen" schedule steps are stored and actually *counted* as steps. Schedule steps are numbered to identify them later on. The first schedule step is also called the "base step." It contains all information necessary to make the MEDL (*Message Descriptor List*, see Chapter 5, Section 5.3.1) for each host. In later schedule steps, additional messages can be added for transmission in previously unused portions of frames. Since the MEDL only contains information about the lengths of the frames, but not their contents, the addition of messages can be done without changing the MEDL.

$^{\textbf{TTX}}$**Plan**

$^{\text{TTX}}$Plan is the cluster design tool for FlexRay clusters. Incremental scheduling is of special interest here, as the Field Bus Exchange Format (FIBEX) [13] is used, and FIBEX also allows us to save just parts of a cluster schedule. Furthermore, FlexRay comprises a static and a dynamic segment, but the concept of schedule steps is not applicable to the dynamic segment.

During FIBEX import, any already existing schedule information is imported first, then the *static* part of the schedule is frozen and the rest of the information is imported. With the command "Make new schedule," this remaining data, including the whole dynamic segment, is included in the schedule. The dynamic segment is *always* scheduled from scratch, regardless of any already existing schedule information. Part of the reason is that the length and the structure of dynamic frames change when messages are added.

$^{\text{TTX}}$Plan adds all schedule increments to its model. When the scheduler is then started to generate a new schedule, it takes into account the original schedule while computing a schedule for the "extended" model. It will not change the global FlexRay configuration, but will eventually allocate additional free slots to hosts and map additional messages to empty spaces in frames. Hosts, subsystems, messages, frames and their associations that were present in the original cluster design remain unchanged. The advantage of this concept is that hosts which are not affected by a change need not be touched. Moreover, a host may support different versions of the schedule by identifying which messages are sent.

Change Management

If, for example, only two hosts A and B need additional messages, only these two must be updated, while all other hosts can remain at the base step of the scheduling. Later, host C might be updated to use the second schedule step, too. Eventually, hosts A, D, and E might get updated to yet another schedule step with additional messages. At runtime, a cluster using incremental scheduling can thus contain hosts with differing schedule steps.

Each schedule step is an extension of the cluster's communication properties. It can place messages into unused parts of already allocated frames or assign yet unused frames to the host and put messages there. When a host has exhausted the spare capacity of its frames, or is known not to want to participate in any further schedule steps, it should be excluded from further schedule steps. The user may then still add increments to other hosts. The dynamic segment is not affected by this exclusion.

To allow for safe interoperation of hosts at various steps of an incremental schedule, each of the hosts participating in a schedule step should send one message per schedule step carrying the schedule-step checksum (e.g., computed by a design tool) which allows for online consistency checks. For a schedule step to be safely usable, the schedule-step checksum sent by the sender must be equal to the schedule-step checksum expected by the receiver.

15.3.2.2 Host Multiplexing

Host Multiplexing is a means to describe the fact that two or more hosts use the same sending slot in different rounds. Although this is a general concept, it is only available for TTP clusters.

A rather simple scenario is given in Figure 15.1. The first three slots are occupied as usual: Each slot is assigned to one node. The last slot is assigned to three nodes, where "Node 3" occupies two rounds, and "Node 4" and "Node 5" each occupy a single round in this four-round schedule.

In the following example scenario, a special kind of host has been designed to be non-periodic and still participate in the multiplexing. It is important to notice that the messages of this host are still periodic! It meets additional requirements like the following:

- One slot (in a schedule of 32 rounds) shall be shared by six hosts.
- Each host shall be assigned one round-slot every 8th round (periodic data).
- In the remaining $4 * 2$ rounds (two per multiplexing period), each host shall be assigned one additional round-slot (event data, higher-level protocols).

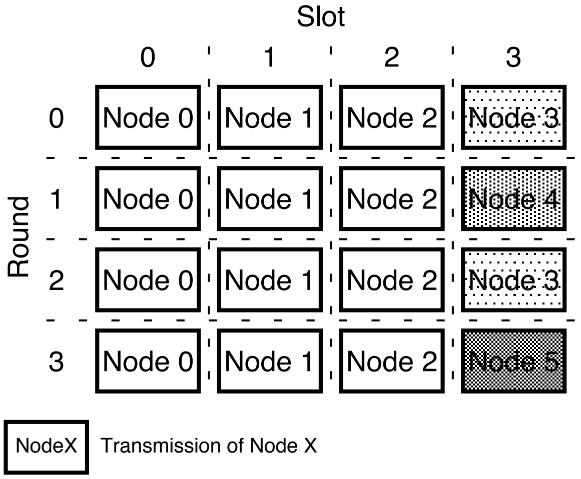


FIGURE 15.1
Multiplexed Slots

- With hosts A to F, the 32 round-slots shall be shared like this (typed in four lines, each representing 8 rounds, for better readability):

```
A B C D E F A B
A B C D E F C D
A B C D E F E F
A B C D E F ? ?
```

- The remaining two round-slots (marked “? ?”) can be assigned to any multiplexing partner.

The pattern required is non-periodic in the sense that transmissions by one multiplexing host are not separated by a constant number of rounds anymore. However, it can still be modeled by assigning *multiple* periods to a single *multiplexing* host (e.g., in the above example, both “mux_periods” 8 and 32 could be assigned to the same host). This type of host is called “MUX_Ghost” (in the following, simply called “ghost”) and has the following properties:

- A ghost behaves like a host in that it can run subsystems in a cluster and can thus send messages. In addition, it must be assigned a “mux_period” and a “mux_round.”
- It is linked to a specific host which implements the subsystems specified for the ghost. (*Note:* A ghost must be linked to the same slot as the linked host.)
- A ghost has no “Host_in_Cluster” link in the object model.

- A ghost has no MEDL.
- The MEDL of a host contains the host's own round-slots ("R_Slot") and the round-slots of all ghosts linked to it.

15.3.2.3 Dynamic Messaging

Dynamic messaging is a concept to support the separation of concerns. One concern is the time, period and data size in which a specific host is permitted to send its data. The other concern is the actual layout and content of the frame being sent. This means that the middleware (e.g., the COM layer) needs to know both "when" and "what" to receive. Hence, it must be configured accordingly. Any time the "what" changes, it needs to be reconfigured.

The general idea — or rather: the requirement — behind dynamic messaging is that the middleware only should know the "when," and consequently only should need to be reconfigured in case of big changes, such as the timing of frames, if at all. Reconfiguration shall not be necessary if a message is added to a "hole" in an existing frame. It definitely shall not be necessary for *all* hosts in the cluster. Dynamic messaging therefore allows us to keep changes local, and to reduce certification efforts.

With dynamic messaging, every message is assigned an ID that is part of the message. It is placed at the beginning of the message, similarly to a frame header, and has a fixed length. With this ID, the embedded software or the COM layer can identify the message within a frame. The obvious disadvantage is that an additional ID per message needs to be transmitted, which requires more bandwidth. The major advantage is that a middleware layer (e.g., the COM layer) does not need any information about the location of a message within a frame. The middleware is able to pack and unpack any message without the communication configuration (MEDL) being modified, too. Allocation is statically predefined, so that overloading of frames cannot occur.

Initially, all hosts get a description of *all* possible messages that exist in the cluster, including their ID, length and other relevant properties for packing and unpacking. Once known, there is no need to update this information, regardless of whether the middleware is transferred to another host, or the message is placed at another position in the frame. Middleware configuration data only needs to be created once, and is the same for all hosts of the cluster. Having host hardware with preloaded and preconfigured middleware on stock becomes feasible, as it can be used right out of the box.

Dynamic messaging can be seen as an alternative to incremental scheduling. While for incremental scheduling, the bin-packing problem needs to be solved for placing messages in frames, and enough room must be reserved for potential future extensions, this is not relevant for dynamic messaging. The layout of the frame is determined at runtime.

15.3.2.4 Scheduling Strategies in $^{\text{TTP}}$Plan

The basic input data for the message scheduler of $^{\text{TTP}}$Plan consists of general cluster information (e.g., cycle durations, transmission speed, topology), information about hosts connected to this cluster and the messages sent by these hosts (e.g., size, period, redundancy).

The message scheduler of $^{\text{TTP}}$Plan is an algorithm to produce a static, cyclic schedule. It is implemented as a heuristic scheduler, or more precisely, as a combination of a list scheduler, followed by an optimization step. The schedule output is basically a set of frames with a specific message allocation and a predefined transmission time instance.

In terms of programming, the message scheduler consists of five steps:

1. Initialization of the scheduler
2. Preparation for the scheduling (including checking the input object model)
3. Scheduling of the messages (including placement of the messages within a frame)
4. Write back the scheduling results to the object model
5. Finish scheduling

Preparation for Scheduling

Before the actual message scheduling takes place, various preparation steps have to be performed inside the message scheduler. This includes increasing the global cluster schedule step and figuring out the number of cluster modes. Usually, there is one user mode and one pseudo mode for TTP startup, but there might be more.

Afterwards, some messages are created that are needed for certain services. Such messages include “RPV messages” for the remote-pin-voting feature, as well as subsystem status messages. Every subsystem that was designed to send its status needs to send such a message. If the cluster allows schedule extensions, special messages carrying schedule step checksums have to be created as well.

Algorithmic Steps

In terms of algorithmic structure and complexity, only the third step from the above list is of interest. It can be broken down further into eight steps. These — basically independent — steps of the message scheduler are described in the order of invocation inside $^{\text{TTP}}$Plan.

1. *Increment the schedule step.* The “scheduled” attribute of all objects is increased by one. This attribute is initially zero if no schedule step has been made so far (base step), and therefore incremented to one. If a schedule of an old, not frozen schedule step exists, this schedule is deleted. All frozen schedule information will be kept.

2. *Create the grid.* This step is only done inside the base step and is skipped for every additional schedule step. The grid is derived from the basic bus parameters like bus speed, the shortest and longest period of messages to be sent and the number of hosts in the cluster. Each cell of the grid represents a round-slot, and an “R_Slot” object is created accordingly. In this step, the number of rounds per cluster cycle is calculated, too.

3. *Schedule messages.*

   (a) Assign one slot to each host, depending on the shortest message period this host wants to use.

   (b) Assign additional slots to hosts according to the user settings regarding reserved bandwidth. With bandwith reservation, the amount of free space within a frame can be influenced, thus facilitating extensions in future schedule steps.

   (c) Determine the “difficulty” of a host by the number of messages, the replica level, and the ID of the host. (The ID is used to obtain a deterministic ordering.)

   (d) For every host, starting with the most difficult one, do the following:

      i. Determine the difficulty of a message in the following order: Channel freedom, redundancy degree, round-delta, round freedom, size and name.

      ii. Assign messages to frames starting with the most difficult message.

      iii. For each message: If there is an available R_Slot, use the R_Slot with “good” round-delta. Otherwise, try to assign a new R_Slot.

      iv. For each slot: Try to balance channels, then try to balance rounds. Slots are not balanced.

4. *Schedule messages in frame.* Place the messages in a specified position inside the frame. There are several options for this placement: The placement can be optimized for data access, leading to messages aligned with byte and word boundaries, as far as possible. It is also possible to specify that a message may be placed in fragments (i.e., not contiguously). A very simple approach is to place one message after the other, in the order they have been added to the frame.

5. *Schedule messages in message boxes.* If message boxes exist, place the messages inside the defined message box depending on alignment, size and ID.

6. *Place I-Frames.*: Place the frames necessary for synchronization of TTP wherever possible. If too few locations can be identified, a warning is issued. In this case, the user may try scheduling with different parameters, or switch over to using X-frames.

7. *Check schedule invariants.* These checks are executed to ensure the consistency of the schedule itself. If an internal error occurs, all schedule information

collected so far will be deleted again. In addition, the schedule signatures and the checksum are computed and set during this check.

15.3.3 Schedule Visualization

The more complex a communication system is, the greater the need for a means to visualize its schedule. It has been shown that increased complexity makes it more difficult to recognize design faults, simply due to a lack of overview. Thus, if the system can be visualized in terms of underlying communication *structures* instead of just pouring out all schedule details over the user, design comprehension is improved [280].

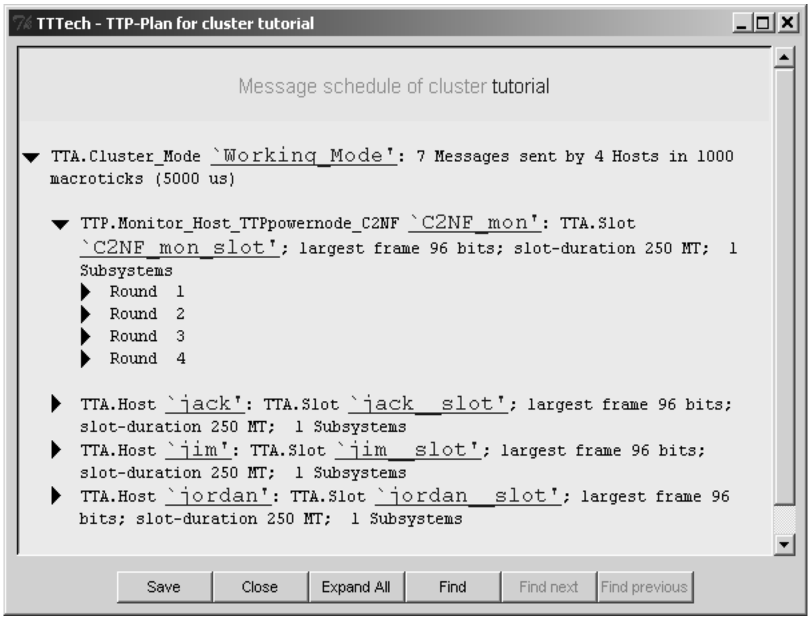
Many characteristics of time-triggered systems — such as their repetitive character (i.e., periodic transmission), predefined "active intervals," the use of state messages for data sharing and highly self-contained components — provide this kind of structure and hence support design comprehension.

For example, the points in time when events in a time-triggered system take place are well-defined. This information can be used to add to an understanding of the system, as the time axis can serve as the basis for conceptual structuring.

On the application level, strictly time-triggered systems just use interfaces based on state messages. This means that the interfaces of all components only consist of a number of state messages that must either be read or written. No other communication or coordination mechanisms are required. As time-triggered systems are of repetitive nature, a component regularly reads the same input messages and then writes the same set of output messages — usually at about equidistant points in time. Only the content of the messages changes, but not the messages themselves.

With these characteristics of a time-triggered system in mind, we can define basically three possibilities for schedule visualization: a *textual* representation (in the following called *schedule browser*), a graphical one (in the following called *schedule viewer* or *schedule editor*) and animation.

While a schedule editor may give a better overview of the whole schedule and eases real "schedule editing" (for example, manually moving frames), a schedule browser may be simpler to use when searching for specific information or wanting to compare certain properties of messages. Animation, although trendy, is not covered here, as we do not consider it a viable solution. In our opinion, it does not satisfy the user's need for interaction (editing) and customized views the way browsers and editors do. Therefore, only examples of these two types are briefly outlined in the following, as they are also implemented in TTTech's readily available cluster design and scheduling tool $^{\text{TTP}}$Plan. $^{\text{TTP}}$Plan can generate a cluster (i.e., message) schedule either from scratch or by extending an existing schedule (*schedule extension*), and provides both textual and graphical schedule editing. Further details can be found in [344].



Message schedule of cluster tutorial

```
▼ TTA.Cluster_Mode `Working_Mode': 7 Messages sent by 4 Hosts in 1000
  macroticks (5000 us)

    ▼ TTP.Monitor_Host_TTPpowernode_C2NF `C2NF_mon': TTA.Slot
      `C2NF_mon_slot'; largest frame 96 bits; slot-duration 250 MT;  1
      Subsystems
      ▶ Round  1
      ▶ Round  2
      ▶ Round  3
      ▶ Round  4

    ▶ TTA.Host `jack': TTA.Slot `jack__slot'; largest frame 96 bits;
      slot-duration 250 MT;  1 Subsystems
    ▶ TTA.Host `jim': TTA.Slot `jim__slot'; largest frame 96 bits;
      slot-duration 250 MT;  1 Subsystems
    ▶ TTA.Host `jordan': TTA.Slot `jordan__slot'; largest frame 96
      bits; slot-duration 250 MT;  1 Subsystems
```

FIGURE 15.2
The Schedule Browser of ^{TTP}Plan

15.3.3.1 The Schedule Browser

The schedule browser of ^{TTP}Plan employs a hierarchical structure, similar to the well-known treeview of other browsers, listing all objects participating in the schedule (hosts, frames, transmission slots). See Figure 15.2 for a screenshot. Each object is displayed as clickable hyperlink, allowing for direct access to the corresponding *object editor*, where the object's attributes can be edited. Expanding an object node in the browser displays the actual timing information of the schedule, e.g., slot durations, frame and message sizes and transmission periods.

A shorter version of the schedule browser, the *schedule summary*, can be useful for a first quick overview. It could be automatically displayed in a design tool right after successful schedule generation, as it is done in ^{TTP}Plan. It only displays the basic data of the generated schedule (number and duration of rounds, transmission speed of messages and frames).

15.3.3.2 The Schedule Editor

In TTP and FlexRay, the communication schedule is based on *rounds* and *slots*. This fact lends itself to a grid-like representation, with the rows corresponding to rounds and the columns corresponding to slots. Each intersection of a row and a column thus represents a *round-slot*, the basic "transmission window" for scheduled data. The grid as a whole displays one cluster cycle in its entirety. Due to the periodic nature of a time-triggered schedule, where only the transmitted contents change, but not the timing behavior, this gives a perfect overview.

In TTP, each transmitting host in the cluster is assigned its own transmission slot. Consequently, the columns automatically also represent the hosts. For FlexRay, an indication which slot is used by which hosts needs to be added.

In a redundant system, i.e., with data being transmitted twice on two different communication channels, each round-slot can be split into two sections to display the frames transmitted on both channels. Vertical alignment of these sections is preferred as the structure of the frames on both channels can be compared quickly, giving an immediate understanding of whether the frames are truly redundant (i.e., have exactly the same structure), or only some messages in the frames are redundant, while others are not.

The schedule editor of $^{\text{TTP}}$Plan is shown in Figure 15.3. It provides drop-down lists to select certain parts of the schedule; this is very helpful when dealing with huge and complex schedules. If a host, frame or message is selected, all occurrences of it are highlighted (as far as the schedule is displayed, that is). For example, selecting a message is useful to see in which slots or rounds it has been scheduled for transmission.

For working with large clusters, the display area of the schedule grid can be set by selecting the desired number of hosts/slots or rounds. On the one hand, this makes the frames larger, easier to see and easier to select with the mouse. On the other hand, it allows us to obtain an overview by viewing all slots and rounds at the same time and to identify "similar" patterns in the communication structure.

As the round-slot fields of the grid may not be large enough (even with a reduced number of visible slots/rounds) to display all relevant information, a "magnifier" function, like the "magnifier window" shown in Figure 15.3, allows the user to view the frames of a selected round-slot — as well as the messages contained in the frames — in a separate window area. In addition, details about the messages (size and timing) are listed below the magnifier window.

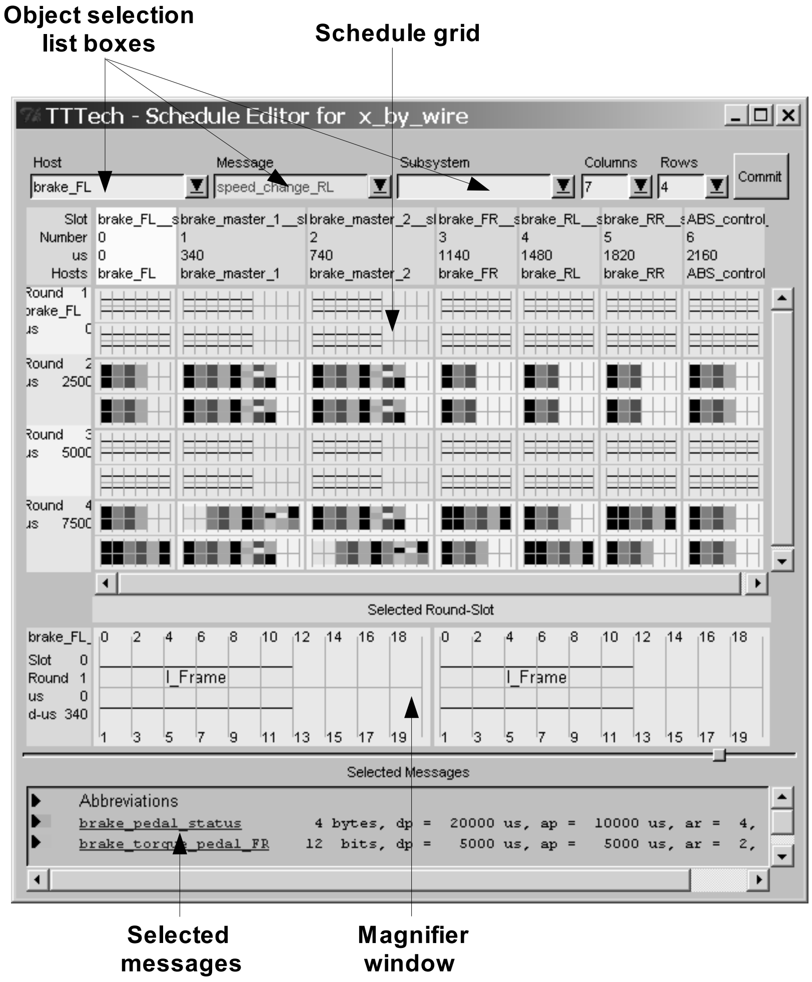
Actual *schedule editing* is best done by drag-and-drop: Drag a message from its current position (frame or round) to another and release it there. This implicitly changes the affected attributes of the message. In this way, one can optimize the current schedule and generate shorter slots, thus allowing for shorter overall rounds. Manual editing also can provide a way out in case the scheduling tool failed to find a feasible schedule.

However, certain actions are prohibited by the schedule editor because they would either violate design constraints or have to be performed prior to rescheduling, i.e., in the scheduling tool itself:

- Drop messages into rounds where their period or phase constraints would be violated

- Drop messages on I-frames (for TTP)

- Move replicated messages to a round-slot where there is not enough space on the other channel (in TTP: where there is an I-frame on one of the channels)



TTTech - Schedule Editor for x_by_wire

Host: brake_FL

Message: speed_change_RL

Subsystem:

Columns: 7

Rows: 4

Commit

Slot	brake_FL__s	brake_master_1__sl	brake_master_2__sl	brake_FR__s	brake_RL__s	brake_RR__s	ABS_control_
Number	0	1	2	3	4	5	6
us	0	340	740	1140	1480	1820	2160
Hosts	brake_FL	brake_master_1	brake_master_2	brake_FR	brake_RL	brake_RR	ABS_control

Round 1
brake_FL
us 0

Round 2
us 2500

Round 3
us 5000

Round 4
us 7500

Selected Round-Slot

brake_FL_
Slot 0
Round 1
us 0
d-us 340

Selected Messages

```
Abbreviations
brake_pedal_status      4 bytes, dp =  20000 us, ap =  10000 us, ar =  4,
brake_torque_pedal_FR   12  bits, dp =   5000 us, ap =   5000 us, ar =  2,
```

FIGURE 15.3
The Schedule Editor of $^{\text{TTP}}$Plan

- Move messages out of their slot (in TTP) or out of the slots the sending host may use (in FlexRay). We consider it bad practice to implicitly change the communication requirements (i.e., who sends what) by editing the schedule. Editing should only refine the timing in detail.

- Move messages within the frame (there should never be a need for this).

15.3.3.3 The Round-Slot Viewer

Similar to a schedule editor, a *round-slot viewer* has a grid-like structure, with the rows representing rounds and the columns representing slots. Each intersection of a row and a column thus represents a *round-slot*. For large schedules, scrolling and limiting the number of displayed items can be useful. After the successful generation of a schedule, one might want to open the round-slot viewer to have a look at the schedule timing.

Like the schedule editor, the round-slot viewer shown in Figure 15.4 provides a magnifier window below the schedule grid. Selecting a round-slot highlights it and also shows it in the magnifier window. At the top of the magnifier window, the slot time is displayed for both channels (first channel above, second one below). The time is split into four parts that are equal for both channels (from left to right):

- **Transmission phase**: The time span needed for transmission of the frames. I-frames and N-frames are displayed in different colors. Overfull N-frames would be displayed in red to highlight them.

- **Post-receive-phase (prp)**: The time span immediately after transmission phase, during which certain services are performed.

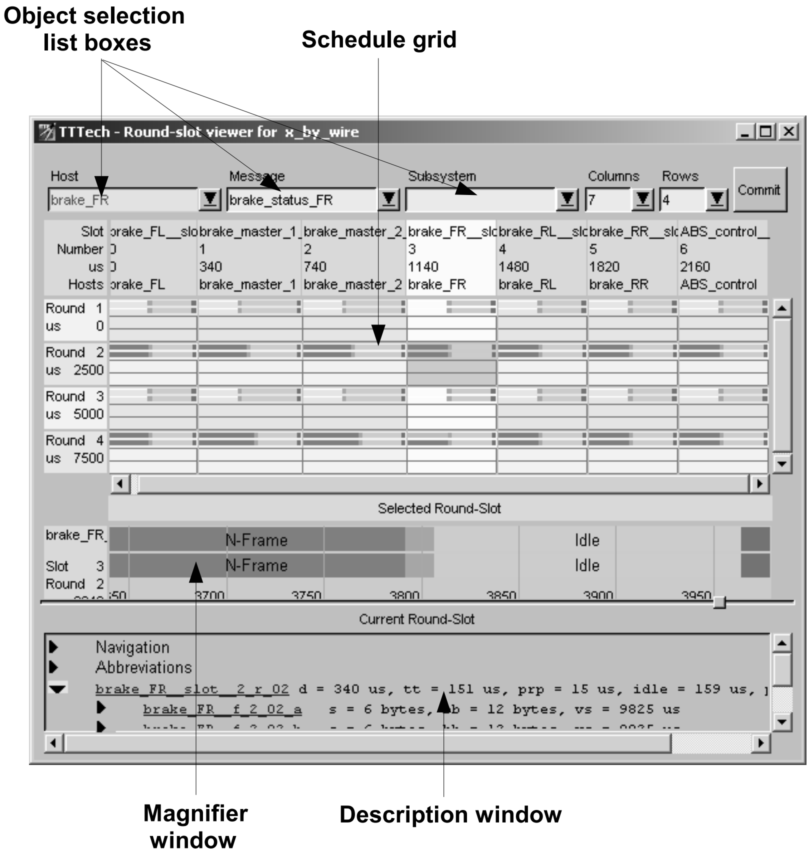
- **Idle time**: This time is needed to stretch the durations of the slots to meet the specified round duration. This idle time is unused bandwidth.

- **Pre-send-phase (psp)**: The time span immediately before action time, during which frame transmission is prepared. The sum of prp, idle time and psp determines the inter-frame gap (IFG). It is limited by the slowest controller in the cluster.

Below the slot time, the user interrupts for both channels are displayed. The magnifier window itself displays additional information about the selected round-slot. Among this information there is the kind of each item in the round-slot, as well as a time grid showing the time from the beginning of the cluster cycle.

15.3.3.4 Visualization of Message Paths

TTEthernet communication, although time-triggered, is not as strict in its structure as TTP. It is not based on rounds and individual sending slots for each device, but rather on “communication links,” i.e., physical connections between sender and receiver, that are basically independent of each other. In contrast to TTP, TTEthernet



TTTech - Round-slot viewer for x_by_wire

Host: brake_FR
Message: brake_status_FR
Subsystem:
Columns: 7
Rows: 4
Commit

Slot	orake_FL__slo	brake_master_1_	brake_master_2_	brake_FR__slc	brake_RL__slc	brake_RR__slc	ABS_control__
Number	0	1	2	3	4	5	6
us	0	340	740	1140	1480	1820	2160
Hosts	orake_FL	brake_master_1	brake_master_2	brake_FR	brake_RL	brake_RR	ABS_control

Round 1 us 0
Round 2 us 2500
Round 3 us 5000
Round 4 us 7500

Selected Round-Slot

brake_FR_ N-Frame Idle
Slot 3 N-Frame Idle
Round 2
:50 3700 3750 3800 3850 3900 3950

Current Round-Slot

```
▶ Navigation
▶ Abbreviations
▼ brake_FR__slot__2_r_02 d = 340 us, tt = 151 us, prp = 15 us, idle = 159 us,
    ▶ brake_FR__f_2_02_a   s = 6 bytes, bb = 12 bytes, vs = 9825 us
```

FIGURE 15.4
The Round-Slot Viewer of ^{TTP}Plan

allows the simultaneous reception and transmission on the same link, as well as simultaneous communication on several links. Therefore, a rigid grid like that of the schedule editor or the round-slot viewer presented above is not the optimal visualization strategy.

The approach presented here is instead based on the communication links and "message paths." A message path denotes the logical path a message takes through the network from the original sender to the last receiver or receivers, including intermediate receiving and resending by one or more switches. Figure 15.5 shows a possible schedule viewer for TTEthernet, based on such a visualization approach. Note that — for simplicity — only strictly time-triggered messages are considered here (i.e., no rate-constrained or best-effort messages).

As usual, the schedule viewer is based on a horizontal time axis. In parallel to it there are the lines representing the communication links. Each link connects two devices, whose names are stated at the left edge of the schedule, above and below the line. The colored rectangles *above* each line are the messages transmitted from the upper to the lower device, the rectangles *below* the line those in the other direction.

The example schedule in Figure 15.5 displays one cluster cycle with a duration of 1 ms, with the first 100 μs being reserved (by design) for special purposes, e.g., clock synchronization. For simplicity, all messages are transmitted once per cluster cycle, i.e., their periods equal the cluster cycle duration.

Following the path of the message `OUT` (gray rectangle) from the main controller to all other devices provides some insight into the way of interpreting the displayed schedule. Moving along the time axis, the following transmissions take place:

1. The *Main Controller* sends `OUT` to switch *sw1* (lower left corner of the schedule).

2. *sw1* takes some time processing `OUT`, hence the gap between the first and the second transmission.

3. *sw1* simultaneously sends `OUT` to the end system IO Node1 and the switch *sw2*.

4. *sw2* takes some time processing `OUT`.

5. *sw2* sends `OUT` to *sw3* just after the processing time.

6. *sw2* simultaneously sends `OUT` to IO Node2 and IO Node3. This transmission takes place later than that to *sw3* because there are some other messages scheduled for transmission in the same direction on these links.

7. *sw3* takes some time processing `OUT`.

8. *sw2* simultaneously sends `OUT` to IO Node4 and IO Node5.

In the same way, the paths of the messages INX and IN_MC_X can be traced from the end systems (IO NodeX) to the main controller (starting at the upper left corner of the schedule).

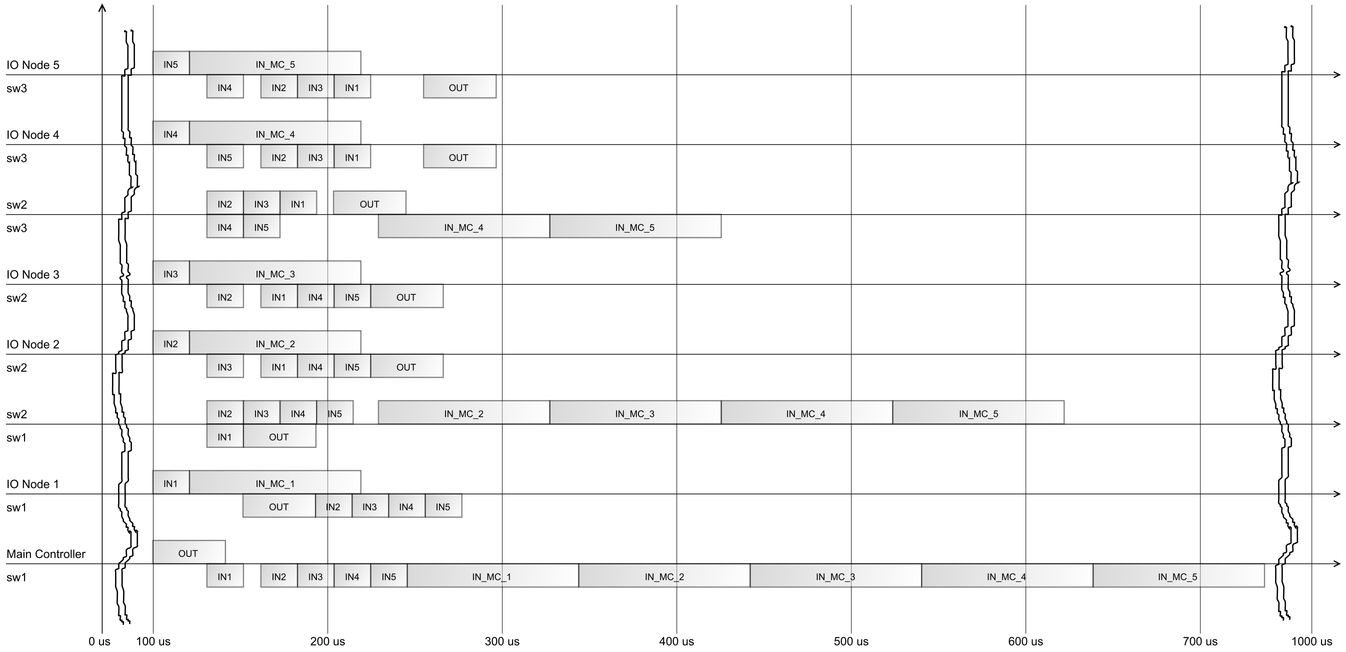


FIGURE 15.5
Illustration of Data Traffic on All Full-Duplex Connections of a TTEthernet Network

Clicking on a message not only highlights all its occurrences, but even shows its message paths as arrows indicating the intra-network communication. User interaction is not viable here, as drawing all arrows for all messages would result in a mess. However, showing the paths of a selected message on request gives the user a quick notion of where processing delays occur and which way the data flows, resulting in a good impression about the latency of this message.

Displaying messages with a transmission period of exactly one cluster cycle — as in the above example — is simple, but a system is not always designed this way. For messages with a higher frequency, i.e., that are transmitted more than once per cluster cycle, additional "depth" of the display is needed. Basically, there are two possibilities to include the periodicity in the displayed schedule:

- To draw one message instance after the other, with the schedule viewer always displaying one whole cluster cycle. The advantage is that this representation is simple. The disadvantage is that the viewing area can become very long and thus will need a lot of scrolling and zooming. By introducing a magnifier or an overview window, navigation in this "wide" schedule representation can be made more comfortable.

- To wrap each link after the shortest period (cf. the TTP schedule editor), which means that each instance of the shortest period starts "at the beginning," i.e., the left edge. The advantage is that the messages with the highest frequency are placed below each other, and messages with periods that are integer multiples of the shortest period are also nicely displayed. But there are at least two disadvantages. First, it is difficult to keep an overview with several links. Second, adding arrows to show the communication paths makes the schedule quite unreadable, as the arrows may cross message instances of interest.

15.4 Holistic Scheduling and Optimization

Applications consist of a set of interacting tasks that communicate through messages. Depending on the functionality, tasks and messages may be time-triggered or event-triggered, or, in certain situations [205], a combination of both. There are many applications where the interaction between the functions is tightly coupled, and design decisions cannot be taken in isolation, they have to be taken considering the complete system, i.e., in a holistic manner. For example, when TT tasks and TT messages are synchronized, the schedule of the tasks has to be constructed at the same time with the message schedule. Also, the worst-case end-to-end delays for ET messages may impact the worst-case response times of ET tasks, and in this case the analysis and optimization of messages has to be considered at the same time with the analysis and optimization of tasks.

In this section, we present an approach to holistic analysis and optimization of

FlexRay-based systems. Although the work here considers FlexRay, the holistic analysis and optimization principles are also valid for other protocols. FlexRay is composed of static (ST) and dynamic (DYN) segments, which are arranged to form a bus cycle that is repeated periodically. The ST segment is similar to TTP, and employs a generalized time-division multiple-access (GTDMA) scheme. The DYN segment of the FlexRay protocol is similar to Byteflight and uses a flexible TDMA (FTDMA) bus access scheme. We propose techniques for determining the timing properties of messages transmitted in the static and the dynamic segments of a FlexRay communication cycle. We first briefly present a static cyclic scheduling technique for TT messages transmitted in the ST segment. Then, we develop a worst-case response time analysis for ET messages sent using the DYN segment, thus providing predictability for messages transmitted in this segment. The analysis techniques for messages are integrated in the context of a holistic schedulability analysis algorithm that computes the worst-case response times of all the tasks and messages in the system.

Such an analysis, while being able to bound the message transmission times on both the ST and DYN segments, represents the first step toward enabling the use of this protocol in a systematic way for time-critical applications. The second step toward an efficient use of FlexRay is concerned with optimization techniques that consider the particular features of an application during the process of finding a FlexRay bus configuration that can guarantee that all time constraints are satisfied.

15.4.1 System Model

We consider architectures consisting of nodes connected by one FlexRay communication channel[2] (see Figure 15.6a). Each processing node connected to a FlexRay bus is composed of two main components: A CPU and a communication controller (see Figure 15.7a) that are interconnected through a two-way controller-host interface (CHI). The controller runs independently of the node's CPU and implements the FlexRay protocol services.

For the systems we are studying, we have made some basic assumptions about the features of a software architecture which runs on the CPU of each node. The main component of the software architecture is a real-time kernel that contains two schedulers, for static cyclic scheduling (SCS) and fixed priority scheduling (FPS), respectively[3] (see Figure 15.6b).

When several tasks are ready on a node, the task with the highest priority is activated, and preempts the other tasks. Let us consider the example in Figure 15.6b, where we have six tasks sharing the same node. Tasks τ_1 and τ_6 are scheduled using SCS, while the rest are scheduled with FPS. The priorities of the FPS tasks are indicated in the figure. The arrival time of a task is depicted with an upward pointing arrow. Under these assumptions, Figure 15.6b presents the worst-case response times of each task. SCS tasks are non-preemptable and their start time is offline fixed in the

[2] FlexRay is a dual-channel bus.

[3] EDF can also be added, as presented by us in [266].

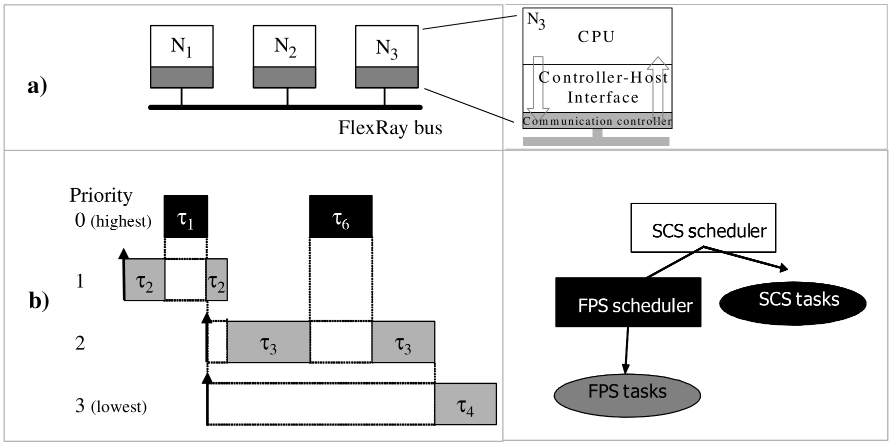


FIGURE 15.6
System Architecture Example

schedule table (they also have the highest priority, denoted with priority level “0” in the figure). FPS tasks can only be executed in the slack of the SCS schedule table.

FPS tasks are scheduled based on priorities. Thus, a higher priority task such as τ_3 preempts a lower priority task such as τ_4. SCS activities are triggered based on a local clock in each processing node. The synchronization of local clocks throughout the system is provided by the communication protocol.

15.4.2 The FlexRay Communication Protocol

In this section, we will describe how messages generated by the CPU reach the communication controller and how they are transmitted on the bus. Let us consider the example in Figure 15.7 where we have three nodes, N_1 to N_3 sending messages $m_a, m_b, \ldots, m_h$ using a FlexRay bus.

In FlexRay, the communication takes place in periodic cycles (Figure 15.7b depicts two cycles of length T_{bus}). Each cycle contains two time intervals with different bus access policies: An ST segment and a DYN segment.[4] We denote with ST_{bus} and DYN_{bus} the length of these segments. In Figure 15.7 there are three static slots for the ST segment. For details on the FlexRay communication protocol, the reader is directed to the FlexRay chapter.

In Figure 15.7, node N_1 has been allocated ST slot 2 and DYN slot 3, N_2 transmits through ST slots 1 and 3 and DYN slots 2 and 4, while node N_3 has DYN slots 1 and 5. For each of these slots, the CHI reserves a buffer that can be written by the CPU and read by the communication controller (these buffers are read by the

[4] The FlexRay bus cycle also contains a *symbol window* and a *network idle time*, but their size does not affect the equations in our analysis. For simplicity, they will be ignored during the examples throughout the section.

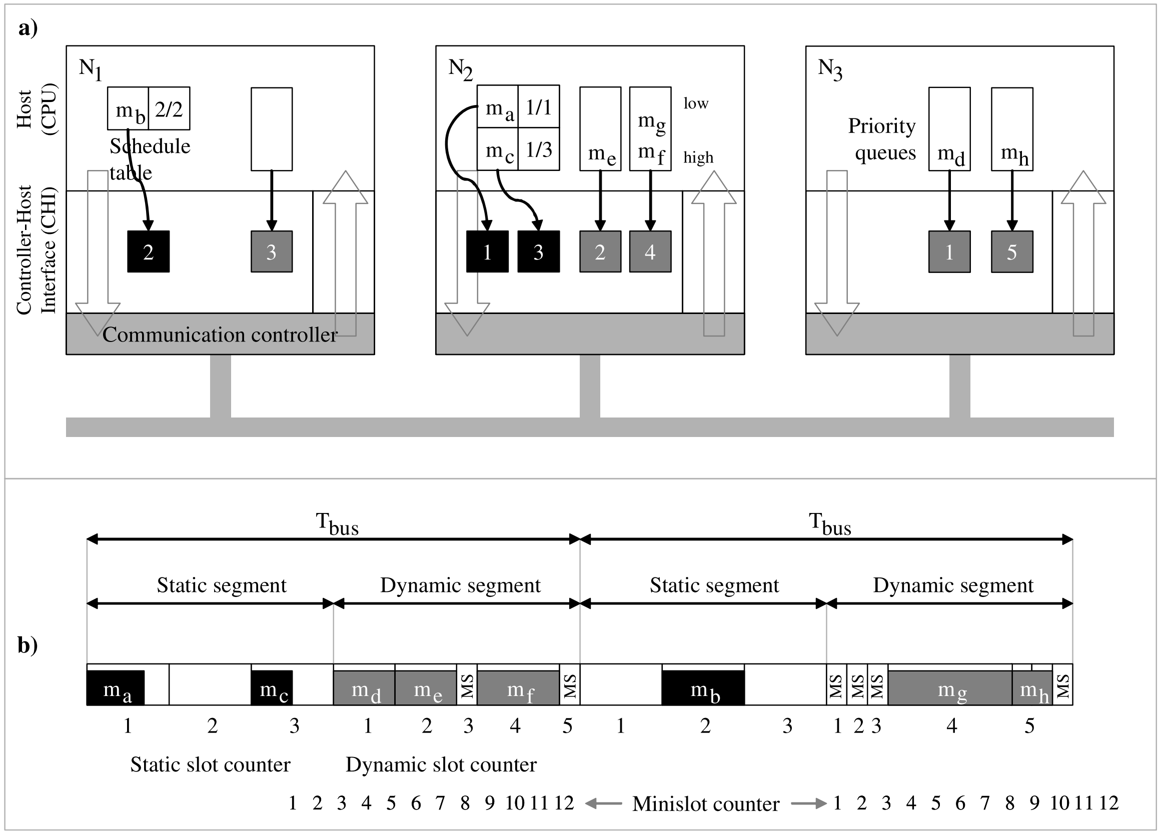


FIGURE 15.7
FlexRay Communication Cycle Example

communication controller *at the beginning* of each slot, in order to prepare the transmission of frames). The associated buffers in the CHI are depicted in Figure 15.7a. We denote with $DYNSlots_{N_p}$ the number of dynamic slots associated to a node N_p (this means that for N_2 in Figure 15.7, $DYNSlots_{N_2}$ has value 2).

We use different approaches for ST and DYN messages to decide which messages are transmitted during the allocated slots. For ST messages, we consider that the CPU in each node holds a schedule table with the transmission times. When the time comes for an ST message to be transmitted, the CPU will place that message in its associated ST buffer of the CHI. For example, ST message m_b sent from node N_1 has an entry "2/2" in the schedule table specifying that it should be sent in the second slot of the second ST cycle.

For the DYN messages, we assume that the designer specifies their *FrameID*. For example, DYN message m_e has the frame identifier "2." While nodes must use distinct *FrameID*s (and consequently distinct DYN slots) in order to avoid bus conflicts, we allow for a node to send different messages using the same DYN *FrameID*.[5] For example, messages m_g and m_f on node N_2 have both *FrameID* 4. If two or more messages with the same frame identifier are ready to be sent in the same bus cycle, a priority scheme is used to decide which message will be sent first. Each DYN message m_i has associated a priority $priority_{m_i}$. Messages with the same *FrameID* will be placed in a local output queue ordered based on their priorities. The message from the head of the priority queue is sent in the current bus cycle. For example, message m_f will be sent before m_g because it has a higher priority.

At the beginning of each communication cycle, the communication controller of a node resets the slot and minislot counters. At the beginning of each communication slot, the controller verifies if there are messages ready for transmission (present in the CHI send buffers) and packs them into frames.[6] In the example in Figure 15.7, we assume that all messages are ready for transmission before the first bus cycle.

Messages selected and packed into ST frames will be transmitted during the bus cycle that is about to start according to the schedule table. For example, in Figure 15.7, messages m_a and m_c are placed into the associated ST buffers in the CHI in order to be transmitted in the first bus cycle. However, messages selected and packed into DYN frames will be transmitted during the DYN segment of the bus cycle only if there is enough time until the end of the DYN segment. Such a situation is verified by comparing if, in the moment the DYN slot counter reaches the value of the *FrameID* for that message, the value of the minislot counter is smaller than a given value *pLatestTx*. The value *pLatestTx* is fixed for each node during the design phase, depending on the size of the largest DYN frame that node will have to send during run-time. For example, in Figure 15.7, message m_h is ready for transmission before the first bus cycle starts, but, after message m_f is transmitted, there is not enough room left in the DYN segment. This will delay the transmission of m_h for the next bus cycle.

[5] This assumption is not part of the FlexRay specification. If messages are not sharing *FrameID*s, this is handled implicitly as a particular case of our analysis.

[6] In this section, we do not address frame-packing [263], and thus assume that one message is sent per frame.

```
GlobalSchedulingAlgorithm()
  1  while TT_ready_list is not empty
  2    select $\tau_{ij}$ from TT_ready_list
  3    if $\tau_{ij}$ is a SCS task then
  4      schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
  5    else // $\tau_{ij}$ is a ST message
  6      schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
  7    end if
  8    update TT_ready_list
  9  end while
end StaticScheduling
schedule_TT_task($\tau_{ij}$, $Node_{\tau_{ij}}$)
  10   find first available time moment $t_s$ after $ASAP_{\tau_{ij}}$
       on $Node_{\tau_{ij}}$
  11   schedule $\tau_{ij}$ after $t_s$ on $Node_{\tau_{ij}}$, so that holistic analysis
       produces minimal worst-case response times
       for FPS tasks and DYN messages
  12   update ASAP for all $\tau_{ij}$ successors
end schedule_TT_task
schedule_ST_msg($\tau_{ij}$, $Node_{\tau_{ij}}$)
  13   find first ST slot($Node_{\tau_{ij}}$) available after $ASAP_{\tau_{ij}}$
  14   schedule $\tau_{ij}$ in that ST slot
  15   update ASAP for all $\tau_{ij}$ successors
end schedule_ST_msg
```

FIGURE 15.8
Global Scheduling Algorithm

15.4.3 Timing Analysis

Given a distributed system based on FlexRay, as described in the previous two sections, the tasks and messages have to be scheduled. For the SCS tasks and ST messages, this means building the schedule tables, while for the FPS tasks and DYN messages we have to determine their worst-case response times.

The problem of finding a schedulable system has to consider two aspects:

1. When performing the schedulability analysis for the FPS tasks and DYN messages, one has to take into consideration the interference from the SCS activities.

2. Among the possible correct schedules for SCS activities, it is important to build one which favors as much as possible the schedulability of FPS activities.

Figure 15.8 presents the global scheduling and analysis algorithm, in which the main loop consists of a list-scheduling based algorithm [62] that iteratively builds the static schedule table with start times for SCS tasks and ST messages.

A ready list (*TT_ready_list*) contains all SCS tasks and ST messages which are ready to be scheduled (they have no predecessors or all their predecessors have already been scheduled). From the ready list, tasks and messages are extracted one by one (Figure 15.8, line 2) to be scheduled on the processor they are mapped to (line 4), or into a static bus-slot associated to that processor on which the sender of the message is executed (line 6), respectively. The priority function which is used to select among ready tasks and messages is a critical path metric, modified by us for the particular goal of scheduling tasks mapped on distributed systems [86]. Let us consider a particular task τ_{ij} selected from the ready list to be scheduled. We consider that $ASAP_{\tau_{ij}}$ is the earliest time moment which satisfies the condition that all preceding activities (tasks or messages) of τ_{ij} are finished (line 10). With only the SCS tasks in the system, the straightforward solution would be to schedule τ_{ij} at the first time moment after $ASAP_{\tau_{ij}}$ when $Node_{\tau_{ij}}$ is free. Similarly, an ST message will be scheduled in the first available ST slot associated with the node that runs the sender task for that message.

As presented by us in [265], when scheduling SCS tasks, one has to take into account the interference they produce on FPS tasks. The function *schedule_TT_task* in Figure 15.8 places a SCS task in the static schedule in such a way that the increase of worst-case response times for FPS tasks is minimized. Such an increase is determined by comparing the worst-case response times of FPS tasks obtained with our holistic schedulability analysis before and after inserting the new SCS task in the schedule [265].

The next subsection presents our solution for computing the worst-case response times of DYN messages, while in Section 15.4.3.2 we will integrate this solution into a holistic schedulability analysis that determines the timing properties of both FPS tasks and DYN messages (which is called in line 11, of *schedule_TT_task* presented in Figure 15.8).

15.4.3.1 Schedulability Analysis of DYN Messages

The worst-case response time R_m of a DYN message m is given by the following equation:

$$R_m(t) = \sigma_m + w_m(t) + C_m, \tag{15.1}$$

where C_m is the message communication time (see Section 15.3.1), σ_m is the longest delay suffered during one bus cycle if the message is generated by its sender task after its slot has passed, and w_m is the worst-case delay caused by the transmission of ST frames and higher priority DYN messages during a given time interval t. For example, in Figure 15.9, we consider that a message m is supposed to be transmitted in the third DYN slot of the bus cycle. The figure presents the case when message m appears during the first bus cycle after the third DYN slot has passed; therefore, the message has to wait σ_m until the next bus cycle starts. In the second bus cycle, the message has to wait for the ST segment and for the first two DYN slots to finish, delay denoted with w_m (that also contains the transmission of a message m' that uses the second DYN slot).

The communication controller decides what message is to be sent on the bus in a certain communication slot *at the beginning* of that slot. As a consequence, in the worst case, a DYN message m is generated by its sender task immediately after the slot with the $FrameID_m$ has started, forcing message m to wait until the next bus cycle starts in order to really start competing for the bus. In conclusion, in the worst

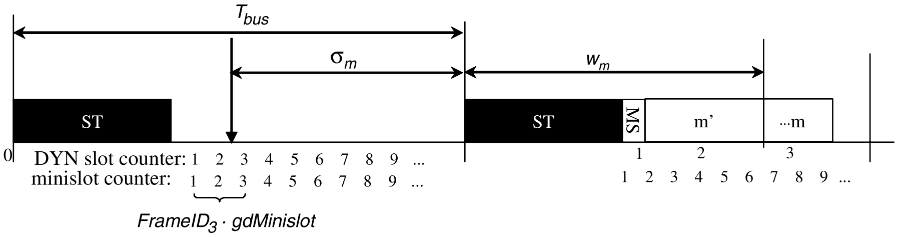

T_{bus}
σ_m
w_m
ST
ST
MS
m'
...m
0
DYN slot counter: 1 2 3 4 5 6 7 8 9 ...
minislot counter: 1 2 3 4 5 6 7 8 9 ...
1 2 3
1 2 3 4 5 6 7 8 9 ...
$FrameID_3 \cdot gdMinislot$

FIGURE 15.9
Response Time of a DYN Message

case, the delay σ_m has the value:

$$\sigma_m = T_{bus} - (ST_{bus} + (FrameID_m - 1) \times gdMinislot), \qquad (15.2)$$

where ST_{bus} is the length of the ST segment.

What is now left to be determined is the value w_m corresponding to the maximum amount of delay on the bus that can be produced by interference from ST frames and DYN messages. We start from the observations that the transmission of a ready DYN message m during the DYN slot $FrameID_m$ can be delayed because of the following causes:

- Local messages with higher priority, that use the same frame identifier as m. We will denote this set of *higher priority local messages* with $hp(m)$. For example, in Figure 15.7a, messages m_g and m_f share *FrameID* 4, thus $hp(m_g) = \{m_f\}$.
- Any messages in the system that can use DYN slots with lower frame identifiers than the one used by m. We will denote this set of messages having *lower frame identifiers* with $lf(m)$. In Figure 15.7a, $lf(m_g) = \{m_d, m_e\}$.
- Unused DYN slots with frame identifiers lower than the one used for sending m (though such slots are unused, each of them still delays the transmission of m for an interval of time equal with the length *gdMinislot* of one minislot); we will denote the set of such minislots with $ms(m)$. Thus, in the example in Figure 15.7b, $ms(m_g) = \{1, 2, 3\}$, and $ms(m_f) = \{3\}$.

Determining the interference of DYN messages in FlexRay is complicated by several factors. Let us consider the example in Figure 15.10, where we have two nodes, N_1 (with *FrameIDs* 1 and 3) and N_2 (with *FrameID* 2), and three messages m_1 to m_3. N_1 sends m_1 and m_3, and N_2 sends message m_2. Messages m_1 and m_2 have *FrameIDs* 1 and 2, respectively. We consider two situations: Figure 15.10a, where m_3 has a separate *FrameID* 3, and Figure 15.10b, where m_3 shares the same *FrameID* 1 with m_1. The values of *pLatestTx* for each node are depicted in the figure.[7]

[7] We use $pLatestTx_m$ to denote $pLatestTx_N$ of the node N sending message m.

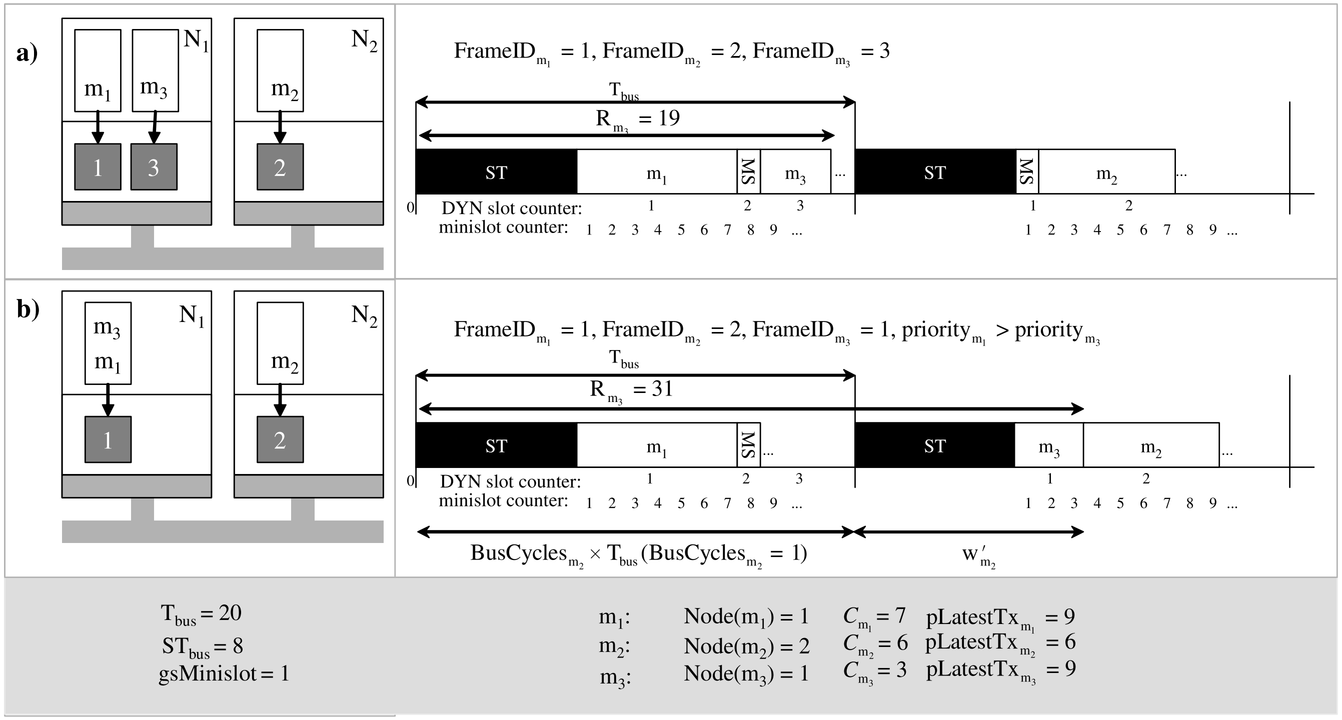

a)
N1
N2
m1
m3
m2
1
3
2
FrameID_m1 = 1, FrameID_m2 = 2, FrameID_m3 = 3
T_bus
R_m3 = 19
ST
m1
MS
m3
...
ST
MS
m2
...
0
DYN slot counter: 1 2 3
minislot counter: 1 2 3 4 5 6 7 8 9 ...
1 2
1 2 3 4 5 6 7 8 9 ...
b)
N1
N2
m3
m1
m2
1
2
FrameID_m1 = 1, FrameID_m2 = 2, FrameID_m3 = 1, priority_m1 > priority_m3
T_bus
R_m3 = 31
ST
m1
MS
...
ST
m3
m2
...
0
DYN slot counter: 1 2 3
minislot counter: 1 2 3 4 5 6 7 8 9 ...
1 2
1 2 3 4 5 6 7 8 9 ...
BusCycles_m2 × T_bus (BusCycles_m2 = 1)
w′_m2
T_bus = 20
ST_bus = 8
gsMinislot = 1
m1: Node(m1) = 1 C_m1 = 7 pLatestTx_m1 = 9
m2: Node(m2) = 2 C_m2 = 6 pLatestTx_m2 = 6
m3: Node(m3) = 1 C_m3 = 3 pLatestTx_m3 = 9

FIGURE 15.10
Transmission Scenarios for DYN Messages

In Figure 15.10a, message m_2, that has a lower FrameID than m_3, cannot be sent immediately after message m_1, because the value of the minislot counter has exceeded the value $pLatestTx_{m_2}$ when the value of the DYN slot counter becomes equal to 2 (hence, m_2 does not fit in this DYN cycle). As a consequence, the transmission of m_2 will be delayed for the next bus cycle. However, since in the moment when the DYN slot counter becomes 3 the minislot counter does not exceed the value $pLatestTx_{m_3}$, message m_3 will fit in the first bus cycle. Thus, a message (m_3 in our case) can be sent before another message with a lower *FrameID*(m_2). Such situations must be accounted for when building the worst-case scenario.

In Figure 15.10b, message m_3 shares the same *FrameID* 1 with m_1 but we consider that it has a lower priority, thus $hp(m_3) = \{m_1\}$. In this case, m_3 is sent in the first DYN slot of the second bus cycle (the first slot of the first cycle is occupied with m_1) and thus will delay the transmission of m_2. In this scenario, we notice that assigning a lower frame identifier to a message does not necessarily reduce the worst-case response time of that message (compare to the situation in Figure 15.10a, where m_3 has $FrameID = 3$).

We next focus on determining the delay $w_m(t)$ in (15.1). The delay produced by all the elements in $hp(m)$, $lf(m)$ and $ms(m)$ can extend to one or more bus cycles:

$$w_m(t) = BusCycles_m(t) \times T_{bus} + w'_m(t), \tag{15.3}$$

where $BusCycles_m(t)$ is the number of bus periods for which the transmission of m is not possible because transmission of messages from $hp(m)$ and $lf(m)$ and because of minislots in $ms(m)$. The delay $w'_m(t)$ denotes now the time that passes, in the last bus cycle, until m is sent, and is measured from the beginning of the bus cycle in which message m is sent until the actual transmission of m starts. For example, in Figure 15.10b, $BusCycles_{m_2} = 1$ and $w'_{m_2}(t) = ST_{bus} + C_{m_3}$. Note that both these terms are functions of time, computed over an analyzed interval t. This means that when computing them we have to take into consideration all the elements in $hp(m)$, $lp(m)$ and $ms(m)$ that can appear during such a given time interval t. Thus, we will consider the multiset $hp(m, t)$ containing all the occurrences over time interval t of elements in $hp(m)$. The number of such occurrences for a message $l \in hp(m)$ is equal to: $\lceil (J_l + t)/T_l \rceil$, where T_l is the period of the message l and J_l is its worst-case jitter (such a jitter is computed as the difference between the worst-case and best-case response times of its sender task s: $J_l = R_s - R_s^b$ [245]). Similarly, $lf(m, t)$ and $ms(m, t)$ consider all the occurrences over t of elements in $lf(m)$ and $ms(m)$, respectively.

The optimal (i.e., exact) solutions for determining the values for $BusCycles_m(t)$ and $w'_m(t)$ are beyond the scope of this section, and are presented in [267]. These, can be intractable for larger problem sizes. Hence, in [267] we have proposed heuristics that quickly compute upper bounds (i.e., pessimistic) values for these terms. Once for any given time interval t we know how to obtain the values $BusCycles(t)$ and $w'_m(t)$, determining the worst-case response time for a message m becomes an iterative process that computes $R_m^k(R_m^{k-1})$, starting from $R_m^0 = C_m$ and finishing when $R_m^k = R_m^{k-1}$.

15.4.3.2 Holistic Schedulability Analysis of FPS Tasks and DYN Messages

As mentioned in Section 15.4.1, the worst-case response times of FPS tasks are influenced on one hand by higher priority FPS tasks, and on the other hand by SCS tasks. The worst-case response time R_{ij} of a FPS task τ_{ij} is determined as presented in [245], and in [265] we have shown how to take into consideration the interference on R_{ij} produced by an existing static schedule. What is important to mention is that R_{ij} depends on jitters of the higher priority tasks and predecessors of τ_{ij}. This means that for all such activities we have to compute the jitter. In the rest of this section, we will only concentrate on the situation when the jitter of a task depends on the arrival time of a message.

According to the analysis of multiprocessor and distributed systems presented in [245], the jitter for a task τ_r that starts execution only after it receives a message m depends on the values of the best-case and worst-case transmission times of that message:

$$J_{\tau_r} = R_m - R_m^b. \tag{15.4}$$

The calculation of the worst-case transmission time R_m of a DYN message m was presented in Section 15.4.3.1. For computing R_m^b we have to identify the best-case scenario of transmitting message m. Such a situation appears when the message becomes ready immediately before the DYN slot with $FrameID_m$ starts, and it is sent during that bus cycle without experiencing any delay from higher priority messages. Thus, the equation for the best-case transmission time of a message is:

$$R_m^b = C_m, \tag{15.5}$$

where C_m is the time needed to send the message m.

We notice from (15.4) that the jitters for activities in the system depend on the values of the worst-case response times, which in turn depend on the values of the jitters [266]. Such a recursive system is solved using a fixed point iteration algorithm in which the initial values for jitters are 0.

According to [245], the worst-case response time calculation of FPS tasks is of exponential complexity and the approach proposed in [245] and also used in [265] is a heuristic with a certain degree of pessimism. The pessimism of the response times calculated by our holistic analysis will, of course, also depend on the quality of the solution for the delay induced by the DYN messages transmitted over FlexRay. The calculation of this delay is our main concern in this section. Therefore, when we speak about optimal and heuristic solutions in this section we refer to the approach used for calculating the $BusCycles_m$ and w'_m (used in the worst-case response times calculation for DYN messages) and not the holistic response time analysis which is based on the heuristics in [245, 265].

For the extension of the analysis to take into account the dual-channel FlexRay bus, we direct the reader to [267].

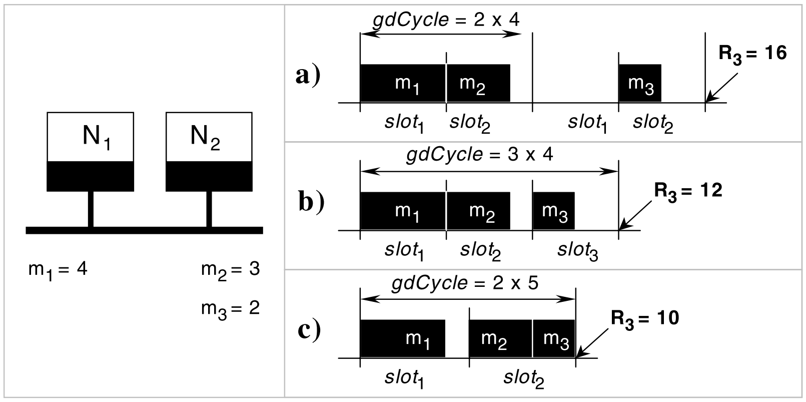


FIGURE 15.11
Optimization of the ST Segment

15.4.4 Bus Access Optimization

The design of a FlexRay bus configuration for a given system consists of a collection of solutions for the following subproblems: (1) determine the length of an ST slot, (2) the number of ST slots, and (3) their assignment to nodes; (4) determine the length of the DYN segment, (5) assign DYN slots to nodes and (6) *FrameID*s to DYN messages.

The choice of a particular bus configuration is extremely important when designing a specific system, since its characteristics heavily influence the global timing properties of the application.

For example, notice in Figure 15.11 how the structure of the ST segment influences the response time of message m_3 (for this example, we ignored the DYN segment). The figure considers a system with two nodes, N_1 that sends message m_1 and N_2 that sends messages m_2 and m_3. The message sizes are depicted in the figure. In the first scenario, the ST segment consists of two slots, *slot*$_1$ used by N_1 and *slot*$_2$ used by N_2. In this situation, message m_3 can be scheduled only during the second bus cycle, with a response time of 16. If the ST segment consists of three slots (Figure 15.11b), with N_2 being allocated *slot*$_2$ and *slot*$_3$, then N_2 is able to send both its messages during the first bus cycle. The configuration in Figure 15.11c consists of only two slots, like in Figure 15.11a. However, in this case the slots are longer, such that several messages can be transmitted during the same frame, producing a faster response time for m_3 (one should notice, however, that by extending the size of the ST slots we delay the reception of message m_1 and m_2).

Similar optimizations can be performed with regard to the DYN segment. Let us consider the example in Figure 15.12, where we have two nodes N_1 and N_2. Node N_1 is transmitting messages m_1 and m_3, while N_2 sends m_2. Figure 15.12 depicts three configuration scenarios, a–c. Table A depicts the frame identifiers for the scenario in Figure 15.12a, while Table B corresponds to Figure 15.12b–c. The length of the

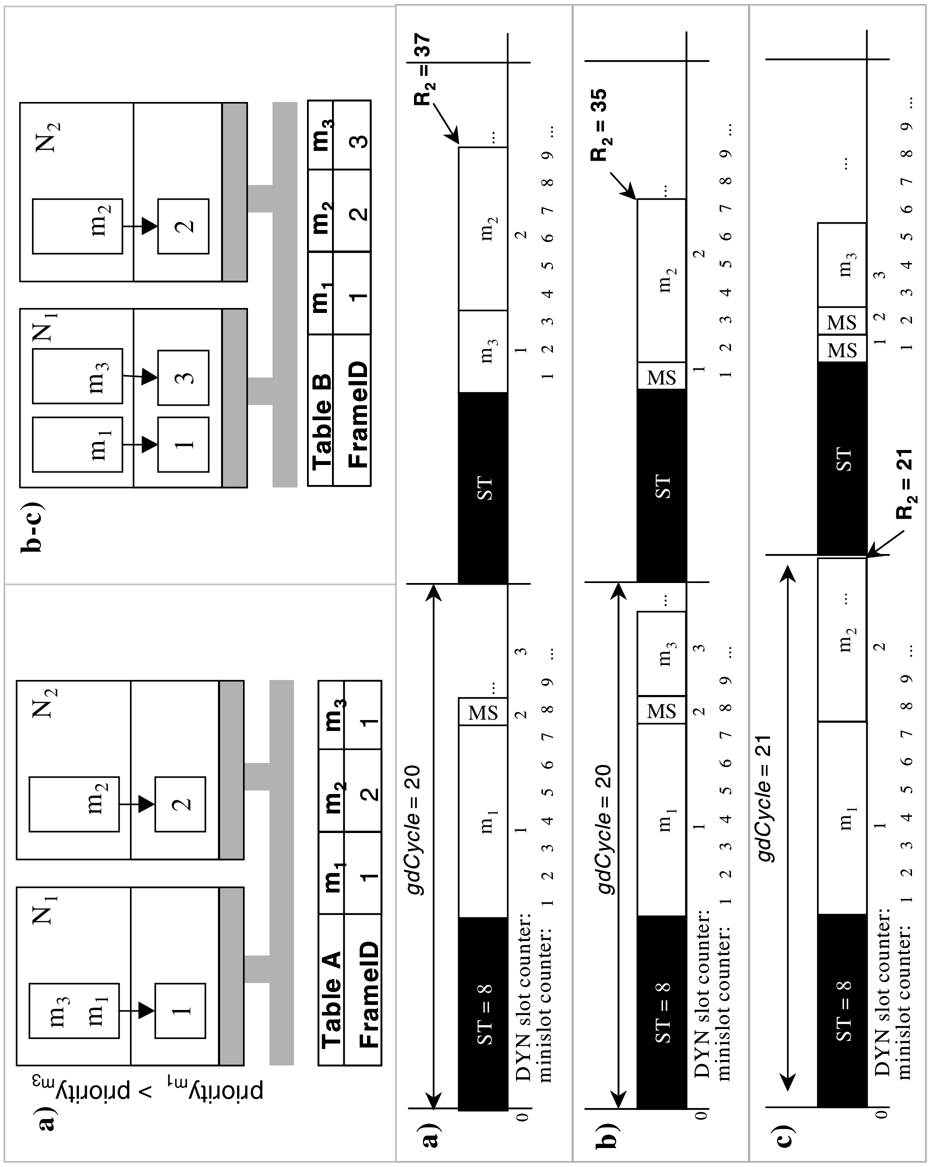


FIGURE 15.12
Optimization of the DYN Segment

```
1  gdNumberOfStaticSlots = max(2, $nodes_{ST}$)
2  gdStaticSlot = max($C_m$), m is an ST message
3  $ST_{bus}$ = gdNumberOfStaticSlots *gdStaticSlot
4  assign one ST slot to each node (round robin)
5  for n = 1 to 64 do
6    gdCycle = $T_{ss}/n$
7    if gdCycle < 16000 $\mu$s then
8      $DYN_{bus}$ = gdCycle − $ST_{bus}$
9      Assign FrameIDs to DYN messages
10     GlobalSchedulingAlgorithm()
11     Compute cost function Cost
12     if Cost < BestCost then save current solution
13   end if
14 end for
```

FIGURE 15.13
Basic Bus Configuration

ST slot has been set to 8. In Figure 15.12a, the length of the DYN segment is not able to accommodate both m_1 and m_2, thus m_2 will be sent during the second bus cycle, after the transmission of m_3 ends. Figure 15.12b and Figure 15.12c depict the same system but with a different allocation of DYN slots to messages (Table B). In Figure 15.12b we notice that m_3, which now does not share the same frame identifier with m_1, can be sent during the first bus cycle, thus m_2 will be transmitted earlier during the second cycle. Moreover, if we enlarge the size of the DYN segment as in Figure 15.12c, then the worst-case response time of m_2 will considerably decrease since it will be sent during the first bus cycle (notice that in this case m_3, having a greater frame identifier than that of m_2, will be sent only during the second cycle).

In order to illustrate the importance of choosing the right bus configuration, we present three approaches for optimizing the bus access such that the schedulability of the system is improved. The first approach builds a relatively straightforward, basic, bus configuration. The other two approaches perform optimization over the basic configuration.

15.4.4.1 The Basic Bus Configuration

In this section, we construct a Basic Bus Configuration (BBC) which is based on analyzing the minimal bandwidth requirements imposed by the application.

The BBC algorithm is presented in Figure 15.13 and it starts by setting the number of ST slots in a bus cycle. The length T_{bus} of the bus cycle is captured by the *gdCycle* protocol parameter. Since each node in the system that generates ST messages needs at least one ST slot, the minimum number of ST slots is $nodes_{ST}$, the number of nodes that send ST messages (line 1). The protocol specification also imposes a minimum limit on the number of ST slots; therefore, even if there are no nodes in the system that are using the ST segment, there should be at least two ST slots during a bus cycle. Next, the size of an ST slot is set so that it can accommodate the largest ST message in the system (line 2). In line 4, the configuration of the ST segment is completed by assigning in a round robin fashion one ST slot to each node that requires one (i.e., in a system with four nodes, where each node is sending in the static segment, the ST segment of the bus cycle will contain four slots; node 1 will use slot 1, node 2 will use ST slot 2, etc.).

When it comes to determining the size of the DYN segment, one has to take into consideration the fact that the period of the bus cycle (*gdCycle*) has to be an integer divisor[8] of the period of the global static schedule (T_{ss}). In addition, the FlexRay protocol specifies that each node implementing a cyclic schedule maintains in the communication controller a counter *vCycleCounter* that has values in the interval $0...63$. This means that during a period of the static schedule there can be at most 64 bus cycles, which leads us to the conclusion that the value of *gdCycle* can be determined by iterating over all possible values for *vCycleCounter* (lines 5–14) and choosing the most favorable solution in terms of system schedulability (line 11). Line 7 introduces a restriction imposed by the FlexRay specification, which limits the maximum bus cycle length to 16 ms. Once the length of the bus cycle is set (line 5), knowing the length ST_{bus} of the ST segment (line 3), we can determine the length DYN_{bus} of the DYN segment (line 8).

At this point, in order to finish the design of the bus configuration, a *FrameID* has to be assigned to each of the DYN messages (and implicitly DYN slots are assigned to the nodes that generate the message). This assignment (line 9) is performed under the following guidelines:

- Each DYN message receives an unique *FrameID*; this is recommended in order to avoid large delays introduced by $hp(m)$. For example, in Figure 15.12, we notice that message m_3 has to wait for an entire *gdCycle* when it shares a frame identifier with the higher priority message m_1(Figure 15.12a), which is not the case when it has its own *FrameID* (Figure 15.12b).

- DYN messages with a higher criticality receive smaller *FrameID*s. This is required in order to reduce, for a given message, the delays produced by $lf(m)$ and $ms(m)$. We capture the criticality of a message m as:

  $$CP_m = D_m - LP_m, \tag{15.6}$$

  where D_m is the deadline of the message and LP_m is the longest path in the task graph from the root to the node representing the communication of message m. A small value of CP_m (higher criticality) indicates that the message should be assigned a smaller *FrameID*.

Once we have defined the structure of the bus cycle, we can analyze the entire system (line 9) by performing the global static scheduling and analysis described in Section 15.4.3. The resulting system is then evaluated using a cost function that captures the schedulability degree of the system (line 10):

[8] We consider that the T_{SS} parameter is slightly adjusted, if necessary.

```
1  for $gdNumberOfStaticSlots = gdNumberOfStaticSlots_m in$ to
   $gdNumberOfStaticSlots_{max}$ do
2    for $gdStaticSlot = gdStaticSlot_{min}$ to $gdStaticSlot_{max}$ step 20 *
     gdBit do
3      Assign ST slots to nodes
4      for $n = 1$ to 64 do
5        $gdCycle = T_{ss}/n$
6        if $gdCycle < 16000\ \mu s$ then
7          $DYN_{bus} = gdCycle - ST_{bus}$
8          do
9            Assign FrameIDs to DYN messages
10           GlobalSchedulingAlgorithm()
11           For all DYN messages, compute $CP_i$
12           Compute cost function $Cost$
13           if $Cost < BestCost$ then save current solution
14         while($BestCost$ unchanged for $max\_iterations$);
15       end if
16     end for
17   end for
18 end for
```

FIGURE 15.14
Greedy Heuristic

$$Cost = \begin{cases} f_1 = \sum_{\tau_{ij}} \max(R_{ij} - D_{ij}, 0), \text{if } f_1 > 0, \\ f_2 = \sum_{\tau_{ij}} (R_{ij} - D_{ij}), \text{if } f_1 = 0 \end{cases} \tag{15.7}$$

where R_{ij} and D_{ij} are the worst-case response times and respectively the deadlines for all the activities τ_{ij} in the system. This function is positive if at least one task or message in the system misses its deadline, and negative if the whole system is schedulable. Its value is used in line 11 when deciding whether the current configuration is the best one encountered so far.

15.4.4.2 Greedy Heuristic

The Basic Bus Configuration (BBC) generated as in the previous section can result in an unschedulable system (the cost function in (15.7) is positive). In this case, additional points in the solution space have to be explored. In Figure 15.14, we present a greedy heuristic that further explores the design space in order to find a schedulable solution.

While for the BBC the number and size of ST slots has been set to the minimum ($gdNumberOfStaticSlots_{min} = \max(2, nodes)$, $gdStaticSlot_{min} = \max(C_m)$), the heuristic explores different alternative values between these minimal values and the maxima imposed by the FlexRay protocol specification. Thus, during a bus cycle there can be at most $gdNumberOfStaticSlots_{max} = 1023$ ST slots, while the size of a ST slot can take at most $gdStaticSlot_{max} = 661$ macroticks. In addition, the payload for a FlexRay frame can increase only in 2-byte increments, which according to the FlexRay specification translates into 20 *gdBit*, where *gdBit* is the time needed for transmitting one bit over the bus (line 2).

The assignment of ST slots (line 3) to nodes is performed, like for the BBC, in a round robin fashion, with the difference that each node can have not only one but a quota of ST slots determined by the ratio of ST messages that it transmits (i.e., a node that sends more ST messages will be allocated more ST slots).

The sizes of the bus cycle and of the DYN segment are assigned in lines 4–16 in a similar way to the BBC algorithm.

However, while for the BBC the allocation of *FrameID*s to DYN messages is based on the estimated criticality (15.6), here we explore several *FrameID* assignment alternatives inside the loop in lines 8–14. We start from an initial assignment as in the BBC after which a global scheduling is performed (line 10). Using the resulted response times, in the next iteration we assign smaller *FrameID*s with priority to those DYN messages m that have a smaller value for $D_m - R_m$, where D_m is the deadline and R_m is the worst-case response time computed by the global scheduling.

15.4.4.3 Simulated Annealing-Based Approach

We have implemented a more exhaustive design space exploration than the one in Section 15.4.4.2, using a Simulated Annealing (SA) [46] approach. While relatively time consuming, this heuristic can be applied if both the BBC and the configuration produced by the greedy approach are unschedulable. Starting from the solution produced by the greedy optimization, the SA based heuristic explores the design space performing the following set of moves:

- *gdNumberOfStaticSlots* is incremented or decremented, inside the allowed limits (when an ST slot is added, it is allocated randomly to a node)
- *gdStaticSlot* is increased or decreased with $20 \times gdBit$, inside the allowed limits
- The assignment of ST slots to nodes is changed by re-assigning a randomly selected ST slot from a node N_1 to another node N_2. We also use in this context a similar transformation that switches the allocation of two ST slots, $FrameID_1$ and $FrameID_2$, used by two nodes N_1 and N_2, respectively
- The assignment of DYN slots to messages is modified by switching the slots used by two DYN messages

In Section 15.4.4.4 we used extensive, time consuming runs with the Simulated Annealing approach, in order to produce a reference point for the evaluation of our greedy heuristic.

15.4.4.4 Evaluation of Bus Optimization Heuristics

In order to evaluate our optimization algorithms, we generated seven sets of 25 applications representing systems of 2 to 7 nodes, respectively. We considered 10 tasks mapped on each node, leading to applications with a number of 20 to 70 tasks. Depending on the mapping of tasks, each such system had up to 60 additional nodes in the application task graph due to the communication tasks. The tasks were grouped

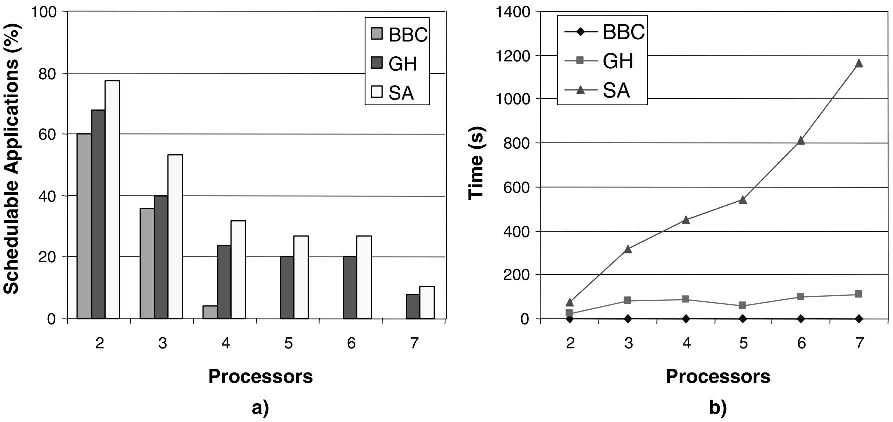


FIGURE 15.15
Evaluation of Bus Optimization Algorithms

in task graphs of five tasks each. Half of the tasks in each system were time triggered and half were event triggered. The execution times were generated in such a way that the utilization on each node was between 30% and 60% (similarly, the message transmission times were generated so that the bus utilization was between 10% and 70%). All experiments were run on an AMD Athlon 2400+ PC.

Figure 15.15 shows the results obtained after running our three algorithms proposed in Section 15.4.4 (BBC—Basic Bus Configuration, GH—Greedy Heuristic, and SA—Simulated Annealing). In Figure 15.15a, we show the percentage of schedulable applications, while in Figure 15.15b, we present the computation times required by each algorithm. One can notice that the BBC approach runs in almost zero time, but it fails to find any schedulable configurations for systems with more than four processors. On the other hand, the other two approaches continue to find schedulable solutions even for larger systems. Moreover, the percentage of schedulable solutions found by the greedy algorithm is comparable with the one obtained with the simulated annealing. Furthermore, the computation time required by the greedy heuristic is several orders of magnitude smaller than the one needed for the extensive runs of simulated annealing.[9]

15.5 Incremental Design

We have briefly introduced the issue of incremental design in Section 15.2. Incremental design has similarities with design for flexibility and scalability. The issue of

[9] Due to the extensive runs with SA, we can assume that the actual percentage of schedulable applications is close to that found by SA.

scalability in time-triggered systems has been investigated in [358], where the authors are interested in generating schedules which (i) allow tasks to increase their WCET without the need for rescheduling and (ii) have idle times distributed periodically to allow future expansion. Haubelt et al. [127] consider the requirement of flexibility as a parameter during design space exploration. Their goal is the generation of an architecture which, at an acceptable cost, is able to implement different applications or variants of a certain application.

In this section, we present an approach for mapping and scheduling of distributed embedded systems for hard real-time applications, aiming at a minimization of the system modification cost. We consider an incremental design process that starts from an already existing system running a set of applications. We are interested in implementing new functionality such that the timing requirements are fulfilled, and the following two requirements are also satisfied: The already running applications are disturbed as little as possible, and there is a good chance that, later, new functionality can easily be added to the resulted system. Thus, we propose a heuristic which finds the set of already running applications which have to be remapped and rescheduled at the same time with mapping and scheduling the new application, such that the disturbance on the running system (expressed as the total cost implied by the modifications) is minimized. Once this set of applications has been determined, we outline a mapping and scheduling algorithm aimed at fulfilling the requirements stated above. The approaches have been evaluated based on extensive experiments using a large number of generated benchmarks.

1. First, we consider mapping and scheduling for hard real-time embedded systems in the context of a realistic communication model based on a time division multiple access (TDMA) protocol as recommended for applications in areas like, for example, automotive electronics [180]. We accurately take into consideration overheads due to communication and consider, during the mapping and scheduling process, the particular requirements of the communication protocol.

2. Next, we have considered the design of distributed embedded systems in the context of an incremental design process as outlined above. This implies that we perform mapping and scheduling of new functionality on a given distributed embedded system, so that certain design constraints are satisfied and, in addition: (a) The already running applications are disturbed as little as possible. (b) There is a good chance that, later, new functionality can easily be mapped on the resulted system.

We propose a new heuristic, together with the corresponding design criteria, which finds the set of old applications which have to be remapped and rescheduled at the same time with mapping and scheduling the new application, such that the disturbance on the running system (expressed as the total cost implied by the modifications) is minimized. Once this set of applications has been determined, mapping and scheduling are performed according to the requirements stated above.

Supporting such a design process is of critical importance for current and future

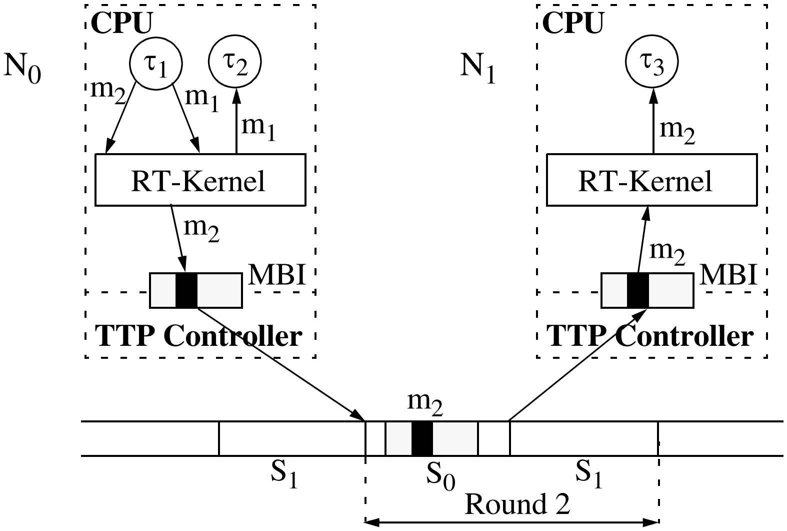


FIGURE 15.16
Message Passing Mechanism

industrial practice, as the time interval between successive generations of a product is continuously decreasing, while the complexity due to increased sophistication of new functionality is growing rapidly. The goal of reducing the overall cost of successive product generations has been one of the main motors behind the, currently very popular, concept of platform-based design [163, 216]. Although, in this section, we are not explicitly dealing with platform-based systems, most of the results are also valid in the context of this design paradigm.

15.5.1 Preliminaries

In this chapter, the concepts of incremental design are investigated in the context of TTP-based systems. However, the techniques presented here are also applicable to other time-triggered protocols.

15.5.1.1 System Architecture

Thus, we consider architectures consisting of nodes connected by a broadcast communication channel. Every node consists of a TTP controller, processor, memory and an I/O interface to sensors and actuators. For the details of TTP, please refer to Chapter 5.

We assume that each node in the architecture has a real-time kernel as its main component. Each kernel has a schedule table that contains all the information needed to take decisions on activation of tasks and each communication controller has a schedule table to decide the transmission of messages.

The message passing mechanism is illustrated in Figure 15.16, where we have three tasks, τ_1 to τ_3. τ_1 and τ_2 are mapped to node N_0 that transmits in slot S_0, and τ_3 is mapped to node N_1 that transmits in slot S_1. Message m_1 is transmitted between τ_1 and τ_2 that are on the same node, while message m_2 is transmitted from τ_1 to τ_3 between the two nodes. We consider that each task has its own memory locations for the messages it sends or receives and that the addresses of the memory locations are known to the kernel through the schedule table.

τ_1 is activated according to the schedule table, and when it finishes it calls the `send` kernel function in order to send m_1, and then m_2. Based on the schedule table, the kernel copies m_1 from the corresponding memory location in τ_1 to the memory location in τ_2. When τ_2 will be activated, it finds the message in the right location. According to our scheduling policy, whenever a receiving task needs a message, the message is already placed in the corresponding memory location. Thus, there is no overhead on the receiving side, for messages exchanged on the same node.

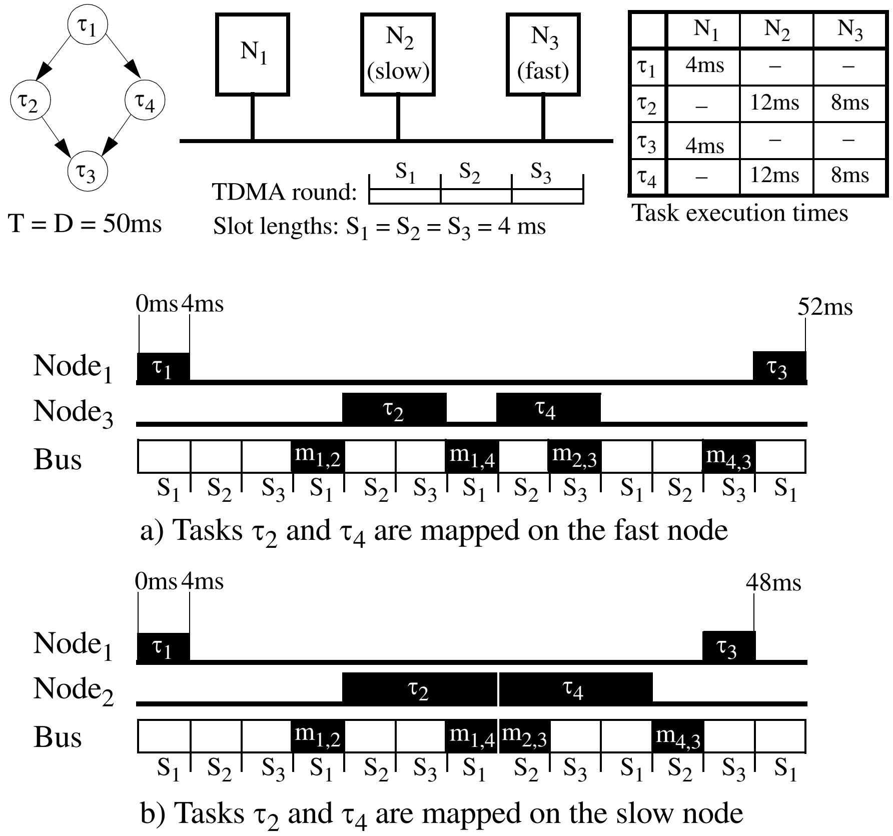
Message m_2 has to be sent from node N_0 to node N_1. At a certain time, known from the schedule table, the kernel transfers m_2 to the TTP controller by packaging it into a frame in the MBI. Later on, the TTP controller knows from its MEDL when it has to take the frame from the MBI, in order to broadcast it on the bus. In our example, the timing information in the schedule table of the kernel and the MEDL is determined in such a way that the broadcasting of the frame is done in the slot S_0 of $Round$ 2. The TTP controller of node N_1 knows from its MEDL that it has to read a frame from slot S_0 of $Round$ 2 and to transfer it into the MBI. The kernel in node N_1 will read the message m_2 from the MBI. When τ_3 will be activated based on the local schedule table of node N_1, it will already have m_2 in its right memory location.

In [260] we presented a detailed discussion concerning the overheads due to the kernel and to every system call. We also presented formulas to derive the worst-case execution delay of a task, taking into account the overhead of the timer interrupt, the worst-case overhead of the task activation and message passing functions.

15.5.1.2 Application Mapping and Scheduling

Considering a system architecture like the one presented earlier, the mapping of a task graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is given by a function $M : \mathcal{V} \rightarrow PE$, where $PE = \{N_1, N_2, .., N_{npe}\}$ is the set of nodes (processing elements). For a task $\tau_i \in \mathcal{V}$, $M(\tau_i)$ is the node to which τ_i is assigned for execution. Each task τ_i can potentially be mapped on several nodes. Let $N_{\tau_i} \subseteq PE$ be the set of nodes to which τ_i can potentially be mapped. For each $N_i \in N_{\tau_i}$, we know the worst-case execution time $t_{\tau_i}^{N_i}$ of task τ_i, when executed on N_i. Messages transmitted between tasks mapped on different nodes are communicated through the bus, in a slot corresponding to the sending node. The maximum number of bits transferred in such a message is also known.

In order to implement an application, represented as a set of task graphs, the designer has to map the tasks to the system nodes and to derive a static cyclic schedule such that all deadlines are satisfied. We first illustrate some of the problems related to mapping and scheduling, in the context of a system based on a TDMA communi-



	N_1	N_2	N_3
τ_1	4ms	–	–
τ_2	–	12ms	8ms
τ_3	4ms	–	–
τ_4	–	12ms	8ms

Task execution times

FIGURE 15.17
Mapping and Scheduling Example

cation protocol, before going on to explore further aspects specific to an incremental design approach.

Let us consider the example in Figure 15.17 where we want to map an application consisting of four tasks τ_1 to τ_4, with a period and deadline of 50 ms. The architecture is composed of three nodes that communicate according to a TDMA protocol, such that N_i transmits in slot S_i. For this example, we suppose that there is no other previous application running on the system. According to the specification, tasks τ_1 and τ_3 are constrained to node N_1, while τ_2 and τ_4 can be mapped on nodes N_2 or N_3, but not N_1. The worst-case execution times of tasks on each potential node and the sequence and size of TDMA slots are presented in Figure 15.17. In order to keep the example simple, we suppose that the message sizes are such that each message fits into one TDMA slot.

We consider two alternative mappings. If we map τ_2 and τ_4 on the faster processor N_3, the resulting schedule length (Figure 15.17a) will be 52 ms which does not meet the deadline. However, if we map τ_2 and τ_4 on the slower processor N_2, the schedule length (Figure 15.17b) is 48 ms, which meets the deadline. Note, that the total traffic on the bus is the same for both mappings and the initial processor load is 0 on both N_2 and N_3. This result has its explanation in the impact of the communication protocol. τ_3 cannot start before receiving messages $m_{2,3}$ and $m_{4,3}$. However, slot S_2 corresponding to node N_2 precedes in the TDMA round slot S_3 on which node N_3 communicates. Thus, the messages which τ_3 needs are available sooner in the case τ_2 and τ_4 are, counter-intuitively, mapped on the slower node.

But finding a valid schedule is not enough if we are to support an incremental design process as discussed in the introduction. In this case, starting from a valid design, we have to improve the mapping and scheduling so that not only the design constraints are satisfied, but also there is a good chance that, later, new functionality can easily be mapped on the resulted system.

To illustrate the role of mapping and scheduling in the context of an incremental design process, let us consider the example in Figure 15.18. For simplicity, we consider an architecture consisting of a single processor. The system is currently running application Ψ (Figure 15.18a). At a particular moment, application $\mathcal{A}_1$ has to be implemented on top of Ψ. Three possible implementation alternatives for $\mathcal{A}_1$ are depicted in Figure 15.18b1, 15.18c1 and 15.18d1. All three are meeting the imposed time constraint for $\mathcal{A}_1$. At a later moment, application $\mathcal{A}_2$ has to be implemented on the system running Ψ and $\mathcal{A}_1$. If $\mathcal{A}_1$ has been implemented as shown in Figure 15.18b1, there is no possibility to map application $\mathcal{A}_2$ on the given system (in particular, there is no time slack available for task τ_7). If $\mathcal{A}_1$ has been implemented as in Figure 15.18c1 or 15.18d1, $\mathcal{A}_2$ can be correctly mapped and scheduled on top of Ψ and $\mathcal{A}_1$. There are two aspects which should be highlighted based on this example:

1. If application $\mathcal{A}_1$ is implemented like in Figure 15.18c1 or 15.18d1, it is possible to implement $\mathcal{A}_2$ on top of the existing system, without performing any modifications on the implementation of previous applications. This could be the case if, during implementation of $\mathcal{A}_1$, the designers have taken into consideration the fact that, in future, an application having the characteristics of $\mathcal{A}_2$ will possibly be added to the system.

2. If $\mathcal{A}_1$ has been implemented like in Figure 15.18b1, $\mathcal{A}_2$ can be added to the system only after performing certain modifications on the implementation of $\mathcal{A}_1$ and/or Ψ. In this case, of course, it is important to perform as few as possible modifications on previous applications, in order to reduce the development costs.

15.5.2 Problem Formulation

As shown in Section 15.5.1, we capture the functionality of a system as a set of applications. An application $\mathcal{A}$ consists of a set of task graphs $\mathcal{G}_i \in \mathcal{A}$. For each task τ_i in a task graph we know the set N_{τ_i} of potential nodes on which it could be mapped and its worst-case execution time on each of these nodes. We also know the maximum number of bits to be transmitted by each message. The underlying architecture is as presented in Section 15.5.1.1. We consider a non-preemptive static cyclic scheduling policy for both tasks and message passing.

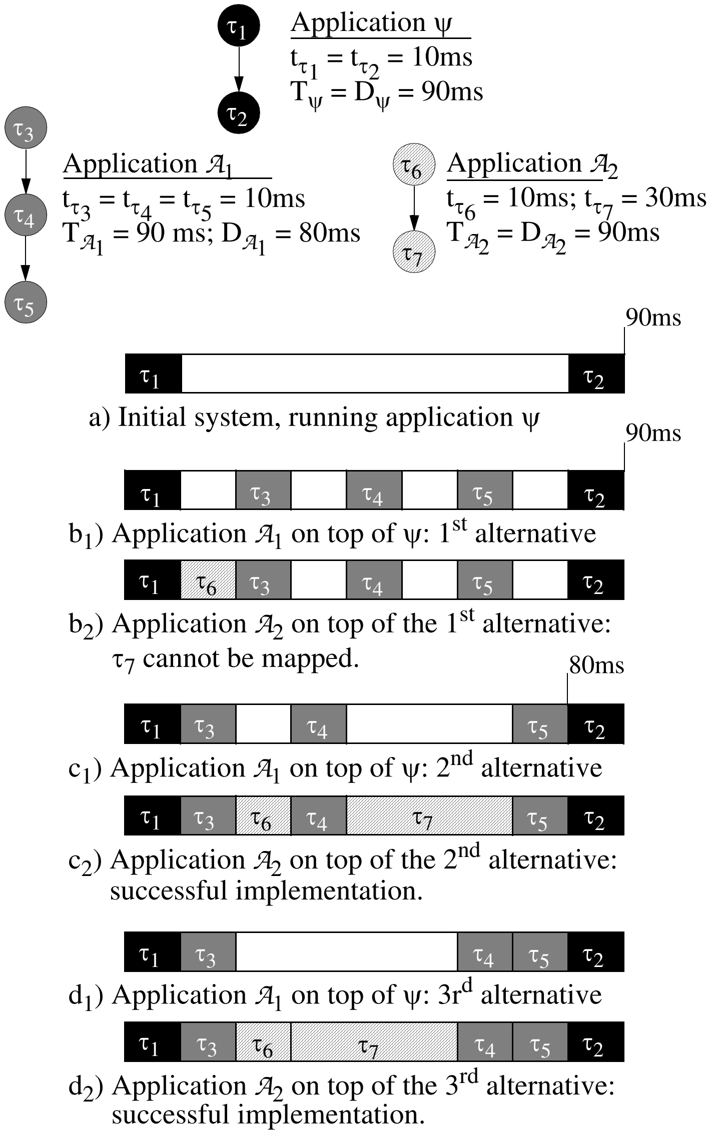
Our goal is to map and schedule an application $\mathcal{A}_{current}$ on a system that already implements a set Ψ of applications, considering the following requirements:

- Requirement a: All constraints on $\mathcal{A}_{current}$ are satisfied and minimal modifications are performed to the implementation of applications in Ψ.
- Requirement b: New applications $\mathcal{A}_{future}$ can be mapped on top of the resulting system.

We illustrate such an incremental design process in Figure 15.19. The product is implemented as a three processor system and its version $N-1$ consists of the set Ψ of two applications (the tasks belonging to these applications are represented as white and black disks, respectively). At the current moment, application $\mathcal{A}_{current}$ is to be added to the system, resulting in version N of the product. However, a new version, $N+1$, is very likely to follow and this fact is to be considered during implementation of $\mathcal{A}_{current}$.[10]

If it is not possible to map and schedule $\mathcal{A}_{current}$ without modifying the implementation of the already running applications, we have to change the scheduling and mapping of some applications in Ψ. However, even with remapping and rescheduling all applications in Ψ, it is still possible that certain constraints are not satisfied. In this case, the hardware architecture has to be changed by, for example, adding a new processor, and the mapping and scheduling procedure for $\mathcal{A}_{current}$ has to be restarted. In this section, we will not further elaborate on the aspect of adding new resources to the architecture, but will concentrate on the mapping and scheduling aspects. Thus, we consider that a possible mapping and scheduling of $\mathcal{A}_{current}$ which satisfies the imposed constraints can be found (with minimizing the modification of

[10] The design process outlined here also applies when $\mathcal{A}_{current}$ is a new version of an application $\mathcal{A}_{old} \in \Psi$. In this case, all the tasks and communications belonging to $\mathcal{A}_{old}$ are eliminated from the running system Ψ, before starting the mapping and scheduling of $\mathcal{A}_{current}$.


Application ψ
$t_{\tau_1} = t_{\tau_2} = 10$ms
$T_\psi = D_\psi = 90$ms
Application $\mathcal{A}_1$
$t_{\tau_3} = t_{\tau_4} = t_{\tau_5} = 10$ms
$T_{\mathcal{A}_1} = 90$ ms; $D_{\mathcal{A}_1} = 80$ms
Application $\mathcal{A}_2$
$t_{\tau_6} = 10$ms; $t_{\tau_7} = 30$ms
$T_{\mathcal{A}_2} = D_{\mathcal{A}_2} = 90$ms
τ_1
τ_2
τ_3
τ_4
τ_5
τ_6
τ_7
90ms
90ms
80ms

a) Initial system, running application ψ

b_1) Application $\mathcal{A}_1$ on top of ψ: 1^{st} alternative

b_2) Application $\mathcal{A}_2$ on top of the 1^{st} alternative: τ_7 cannot be mapped.

c_1) Application $\mathcal{A}_1$ on top of ψ: 2^{nd} alternative

c_2) Application $\mathcal{A}_2$ on top of the 2^{nd} alternative: successful implementation.

d_1) Application $\mathcal{A}_1$ on top of ψ: $3r^{d}$ alternative

d_2) Application $\mathcal{A}_2$ on top of the 3^{rd} alternative: successful implementation.

FIGURE 15.18
Application $\mathcal{A}_2$ Implemented on Top of Ψ and $\mathcal{A}_1$

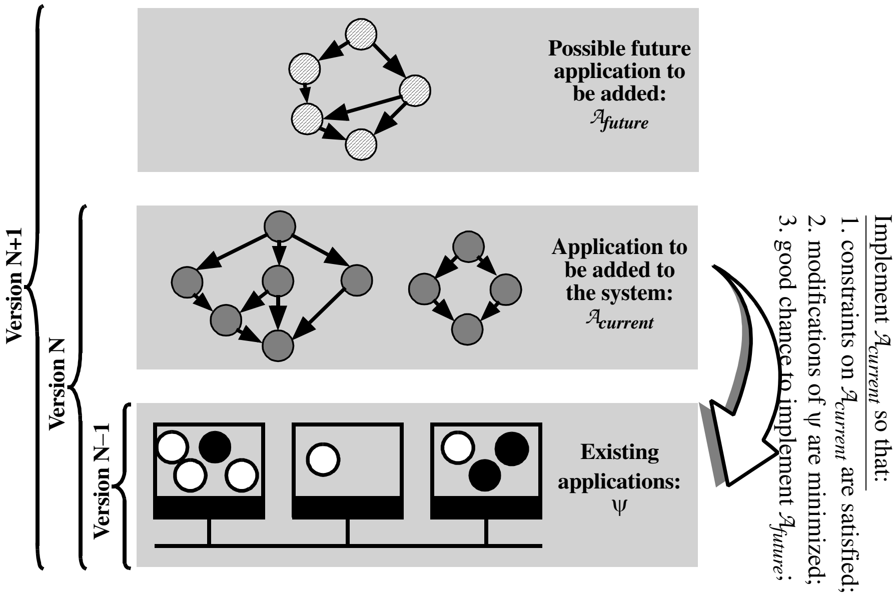

Possible future application to be added: $\mathcal{A}_{future}$
Application to be added to the system: $\mathcal{A}_{current}$
Existing applications: ψ
Version N+1
Version N
Version N–1
Implement $\mathcal{A}_{current}$ so that:
1. constraints on $\mathcal{A}_{current}$ are satisfied;
2. modifications of ψ are minimized;
3. good chance to implement $\mathcal{A}_{future}$;

FIGURE 15.19
Incremental Design Process

the already running applications), and this solution has to be further improved in order to facilitate the implementation of future applications.

In order to achieve our goal, we need certain information to be available concerning the set of applications Ψ as well as the possible future applications $\mathcal{A}_{future}$. What exactly we have to know about these applications will be discussed in Section 15.5.3. In Section 15.5.4 we then introduce the quality metrics which will allow us to give a more rigorous formulation of the problem we are going to solve.

The tasks in application $\mathcal{A}_{current}$ can interact with the previously mapped applications Ψ by reading messages generated on the bus by tasks in Ψ. In this case, the reading task has to be synchronized with the arrival of the message on the bus, which is easy to model as an additional time constraint on the particular receiving task. This constraint is then considered (as any other deadline) during scheduling of $\mathcal{A}_{current}$.

15.5.3 Characterizing Existing and Future Applications

15.5.3.1 Characterizing the Already Running Applications

To perform the mapping and scheduling of $\mathcal{A}_{current}$, the minimum information needed, concerning the already running applications Ψ, consists of the local schedule tables for each processor node. Thus, we know the activation time for each task previously mapped on the respective node and its worst-case execution time. As for messages, their length as well as their place in the particular TDMA frame are known.

If the initial attempt to schedule and map $\mathcal{A}_{current}$ does not succeed, we have to modify the schedule and, possibly, the mapping of applications belonging to Ψ, in the hope of finding a valid solution for $\mathcal{A}_{current}$. The goal is to find that minimal modification to the existing system which leads to a correct implementation of $\mathcal{A}_{current}$. In our context, such a minimal modification means remapping and/or rescheduling a subset Ω of the old applications, $\Omega \subseteq \Psi$, so that the total cost of re-implementing Ω is minimized.

Remapping and/or rescheduling a certain application $\mathcal{A}_i \in \Psi$ can trigger the need to also perform modifications of one or several other applications because of, for example, the dependencies between tasks belonging to these applications. In order to capture such dependencies between the applications in Ψ, as well as their modification costs, we have introduced a representation called the *application graph*. We represent a set of applications as a directed acyclic graph $G(V, E)$, where each node $\mathcal{A}_i \in V$ represents an application. An edge $e_{ij} \in E$ from $\mathcal{A}_i$ to $\mathcal{A}_j$ indicates that any modification to $\mathcal{A}_i$ would trigger the need to also remap and/or reschedule $\mathcal{A}_j$, because of certain interactions between the applications.[11] Each application in the graph has an associated attribute specifying if that particular application is allowed to be modified or not (in which case, it is called frozen). To those nodes $\mathcal{A}_i \in V$ representing modifiable applications, the designer has associated a cost $R_{\mathcal{A}_i}$ of re-implementing $\mathcal{A}_i$. Given a subset of applications $\Omega \subseteq \Psi$, the total cost of modifying the applications in $\mathcal{A}$ is:

$$R(\Omega) = \sum_{\mathcal{A}_i \in \Omega} R_{\mathcal{A}_i}. \tag{15.8}$$

Modifications of an already running application can only be performed if the task graphs corresponding to that application, as well as the related deadlines (which have to be satisfied also after remapping and rescheduling), are available. However, this is not always the case, and in such situations that particular application has to be considered frozen.

In Figure 15.20, we present the graph corresponding to a set of 10 applications. Applications $\mathcal{A}_6$, $\mathcal{A}_8$, $\mathcal{A}_9$ and $\mathcal{A}_{10}$, depicted in black, are frozen: No modifications are possible to them. The rest of the applications have the modification cost $R_{\mathcal{A}_i}$ depicted on their left. $\mathcal{A}_7$ can be remapped/rescheduled with a cost of 20. If $\mathcal{A}_4$ is to be re-implemented, this also requires the modification of $\mathcal{A}_7$, with a total cost of 90. In the case of $\mathcal{A}_5$, although not frozen, no remapping/rescheduling is possible as it would trigger the need to modify $\mathcal{A}_6$, which is frozen.

To each application $\mathcal{A}_i \in V$ the designer has associated a cost $R_{\mathcal{A}_i}$ of re-implementing $\mathcal{A}_i$. Such a cost can typically be expressed in man-hours needed to perform retesting of $\mathcal{A}_i$ and other tasks connected to the remapping and rescheduling of the application. If an application is remapped or rescheduled, it has to be validated again. Such a validation phase is very time consuming. In the automotive industry, for example, the time-to-market in the case of the powertrain unit is 24 months. Out

[11] If a set of applications has a circular dependence, such that the modification of any one implies the remapping of all the others in that set, the set will be represented as a single node in the graph.

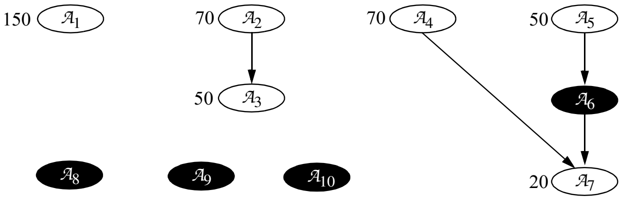


FIGURE 15.20
Characterizing the Set of Already Running Applications

of these, five months, representing more than 20%, are dedicated to validation. In the case of the telematic unit, the time to market is less than one year, while the validation time is two months [291]. However, if an application is not modified during implementation of new functionality, only a small part of the validation tasks have to be re-performed (e.g., integration testing), thus reducing significantly the time-to-market, at no additional hardware or development cost.

How to concretely perform the estimation of the modification cost related to an application is beyond the topic of this section. Several approaches to cost estimation for different phases of the software life-cycle have been elaborated and are available in the literature [75, 271]. One of the most influential software cost models is the Constructive Cost Model (COCOMO) [37]. Such estimations can be used by the designer as the cost metrics assigned to the nodes of an application graph.

In general, it can be the case that several alternative costs are associated to a certain application, depending on the particular modification performed. Thus, for example, we can have a certain cost if tasks are only rescheduled, and another one if they are also remapped on an alternative node. For different modification alternatives considered during design space exploration, the corresponding modification cost has to be selected. In order to keep the discussion reasonably simple, we present the case with one single modification cost associated to an application. However, the generalization for several alternative modification costs is straightforward.

15.5.3.2 Characterizing Future Applications

What do we suppose to know about the family $\mathcal{A}_{future}$ of applications which do not exist yet? Given a certain limited application area (e.g., automotive electronics), it is not unreasonable to assume that, based on the designers' previous experience, the nature of expected future functions to be implemented, profiling of previous applications, available incomplete designs for future versions of the product, etc., it is possible to characterize the family of applications which possibly could be added to the current implementation. This is an assumption which is basic for the concept of incremental design. Thus, we consider that, with respect to the future applications, we know the set $S_t = \{t_{min}, ...t_i, ...t_{max}\}$ of possible worst-case execution times for tasks, and the set $S_b = \{b_{min}, ...b_i, ...b_{max}\}$ of possible message sizes. We also assume that over these sets we know the distributions of probability $f_{S_t}(t)$ for $t \in S_t$ and $f_{S_b}(b)$ for $b \in S_b$. For example, we might have predicted possible worst-case execution times of different tasks in future applications $S_t = \{50, 100, 200, 300, 500\ ms\}$. If there is a higher probability of having tasks of 100 ms, and a very low probability of having tasks of 300 ms and 500 ms, then our distribution function $f_{S_t}(t)$ could look like this: $f_{S_t}(50) = 0.20$, $f_{S_t}(100) = 0.50$, $f_{S_t}(200) = 0.20$, $f_{S_t}(300) = 0.05$, and $f_{S_t}(500) = 0.05$.

Another piece of information is related to the period of task graphs which could be part of future applications. In particular, the smallest expected period T_{min} is assumed to be given, together with the expected necessary processor time t_{need}, and bus bandwidth b_{need}, inside such a period T_{min}. As will be shown later, this information is treated in a flexible way during the design process and is used in order to provide a fair distribution of available resources.

The execution times in S_t, as well as t_{need}, are considered relative to the slowest node in the system. All the other nodes are characterized by a speedup factor relative to this slowest node. A normalization with these factors is performed when computing the metrics C_1^{τ} and C_2^{τ} introduced in the following section.

15.5.4 Quality Metrics and Objective Function

A designer will be able to map and schedule an application $\mathcal{A}_{future}$ on top of a system implementing Ψ and $\mathcal{A}_{current}$ only if there are sufficient resources available. In our case, the resources are processor time and the bandwidth on the bus. In the context of a non-preemptive static scheduling policy, having free resources translates into having free time slots on the processors and having space left for messages in the bus slots. We call these free slots of available time on the processor or on the bus, *slack*. It is to be noted that the total quantity of computation and communication power available on our system after we have mapped and scheduled $\mathcal{A}_{current}$ on top of Ψ is the same regardless of the mapping and scheduling policies used. What depends on the mapping and scheduling strategy is the distribution of slacks along the time line and the size of the individual slacks. It is exactly this size and distribution of the slacks that characterizes the quality of a certain design alternative from the point of view of flexibility for future upgrades. In this section, we introduce two criteria in order to reflect the degree to which one design alternative meets the requirement (b) presented in Section 15.5.2. For each criterion, we provide metrics which quantify the degree to which the criterion is met. The first criterion reflects how well the resulted slack sizes fit to a future application, and the second criterion expresses how well the slack is distributed in time.

15.5.4.1 Slack Sizes (the first criterion)

The slack sizes resulted after implementation of $\mathcal{A}_{current}$ on top of Ψ should be such that they best accommodate a given family of applications $\mathcal{A}_{future}$, characterized by the sets S_t, S_b and the probability distributions f_{S_t} and f_{S_b}, as outlined in Section 15.5.3.2.

Let us go back to the example in Figure 15.18 where $\mathcal{A}_1$ is what we now call

$\mathcal{A}_{current}$, while $\mathcal{A}_2$, to be later implemented on top of Ψ and $\mathcal{A}_1$, is $\mathcal{A}_{future}$. This $\mathcal{A}_{future}$ consists of the two tasks τ_6 and τ_7. It can be observed that the best configuration from the point of view of accommodating $\mathcal{A}_{future}$, taking into consideration only slack sizes, is to have a contiguous slack after implementation of $\mathcal{A}_{current}$ (Figure 15.18d1). However, in reality, it is almost impossible to map and schedule the current application such that a contiguous slack is obtained. Not only is it impossible, but it is also undesirable from the point of view of the second design criterion, to be discussed next. However, as we can see from Figure 15.18b1, if we schedule $\mathcal{A}_{current}$ such that it fragments the slack too much, it is impossible to fit $\mathcal{A}_{future}$ because there is no slack that can accommodate task τ_7. A situation such as the one depicted in Figure 15.18c1 is desirable, where the resulted slack sizes are adapted to the characteristics of the $\mathcal{A}_{future}$ application.

In order to measure the degree to which the slack sizes in a given design alternative fit the future applications, we provide two metrics, C_1^{τ} and C_1^{m}. C_1^{τ} captures how much of the largest future application which theoretically could be mapped on the system can be mapped on top of the current design alternative. C_1^{m} is similar relative to the slacks in the bus slots.

How does the largest future application which theoretically could be mapped on the system look like? The total processor time and bus bandwidth available for this largest future application is the total slack available on the processors and bus, respectively, after implementing $\mathcal{A}_{current}$. Process and message sizes of this hypothetical largest application are determined knowing the total size of the available slack, and the characteristics of the future applications as expressed by the sets S_t and S_b, and the probability distributions f_{S_t} and f_{S_b}. Let us consider, for example, that the total slack size on the processors is 2800 ms and the set of possible worst-case execution times is $S_t = \{50, 100, 200, 300, 500\ ms\}$. The probability distribution function f_{S_t} is defined as follows: $f_{S_t}(50) = 0.20$, $f_{S_t}(100) = 0.50$, $f_{S_t}(200) = 0.20$, $f_{S_t}(300) = 0.05$ and $f_{S_t}(500) = 0.05$. Under these circumstances, the largest hypothetical future application will consist of 20 tasks: 10 tasks (half of the total, $f_t(100) = 0.50$) with a worst-case execution time of 100 ms, 4 tasks with 50 ms, 4 with 200 ms, one with 300 and one with 500 ms.

After we have determined the number of tasks of this largest hypothetical $\mathcal{A}_{future}$ and their worst-case execution times, we apply a *bin-packing algorithm* [215] using the *best-fit policy* in which we consider tasks as the objects to be packed, and the available slacks as containers. The total execution time of tasks which are left unpacked, relative to the total execution time of the whole task set, gives the C_1^{τ} metric. The same is the case with the metric C_1^{m}, but applied to message sizes and available slacks in the bus slots.

Let us consider the example in Figure 15.18 and suppose a hypothetical $\mathcal{A}_{future}$ consisting of two tasks like those of application $\mathcal{A}_2$. For the design alternatives in Figure 15.18c1 and 15.18d1, $C_1^{\tau} = 0\%$ (both alternatives are perfect from the point of view of slack sizes). For the alternative in Figure 15.18b1, however, $C_1^{\tau} = 30/40 = 75\%$ the worst-case execution time of τ_7 (which is left unpacked) relative to the total execution time of the two tasks.

15.5.4.2 Distribution of Slacks (the second criterion)

In the previous section, we defined a metric which captures how well the sizes of the slacks fit a possible future application. A similar metric is needed to characterize the distribution of slacks over time.

Let τ_i be a task with period T_{τ_i} that belongs to a future application, and $M(\tau_i)$ the node on which τ_i will be mapped. The worst-case execution time of τ_i is $t_{\tau_i}^{M(\tau_i)}$. In order to schedule τ_i, we need a slack of size $t_{\tau_i}^{M(\tau_i)}$ that is available periodically, within a period T_{τ_i}, on processor $M(\tau_i)$. If we consider a group of tasks with period T, which are part of $\mathcal{A}_{future}$, in order to implement them, a certain amount of slack is needed which is available periodically, with a period T, on the nodes implementing the respective tasks.

During implementation of $\mathcal{A}_{current}$, we aim for a slack distribution such that the future application with the smallest expected period T_{min} and with the necessary processor time t_{need}, and bus bandwidth b_{need}, can be accommodated (see Section 15.5.3.2).

Thus, for each node, we compute the minimum periodic slack, inside a T_{min} period. By summing these minima, we obtain the slack which is available periodically to $\mathcal{A}_{future}$. This is the C_2^{τ} metric. The C_2^m metric characterizes the minimum periodically available bandwidth on the bus and it is computed in a similar way.

In Figure 15.21 we consider an example with $T_{min} = 120\ ms$, $t_{need} = 90\ ms$ and $b_{need} = 65\ ms$. The length of the schedule table of the system implementing Ψ and $\mathcal{A}_{current}$ is 360 ms (in Section 15.5.5 we will elaborate on the length of the global schedule table). Thus, we have to investigate three periods of length T_{min} each. The system consists of three nodes. Let us consider the situation in Figure 15.21a. In the first period, $Period$ 0, there are 40 ms of slack available on $Node_1$, in the second period 80 ms, and in the third period no slack is available on $Node_1$. Thus, the total slack a future application of period T_{min} can use on $Node_1$ is $min(40, 80, 0) = 0\ ms$. Neither can $Node_2$ provide slack for this application, as in $Period$ 1 there is no slack available. However, on $Node_3$ there are at least 40 ms of slack available in each period. Thus, with the configuration in Figure 15.21a we have $C_2^{\tau} = 20\ ms$, which is not sufficient to accommodate $t_{need} = 90\ ms$. The available periodic slack on the bus is also insufficient: $C_2^m = 60\ ms < b_{need}$. However, in the situation presented in Figure 15.21b, we have $C_2^{\tau} = 120\ ms > t_{need}$, and $C_2^m = 90\ ms > b_{need}$.

15.5.4.3 Objective Function and Exact Problem Formulation

In order to capture how well a certain design alternative meets the requirement (b) stated in Section 15.5.2, the metrics discussed before are combined in an objective function, as follows:

$$C = w_1^{\tau}(C_1^{\tau})^2 + w_1^m(C_1^m)^2 + w_2^{\tau}\max(0, t_{need} - C_2^{\tau}) + w_2^m \max(0, b_{need} - C_2^m) \tag{15.9}$$

where the metric values introduced in the previous section are weighted by the constants w_1^τ, w_2^τ, w_1^m and w_2^m. Our mapping and scheduling strategy will try to minimize this function. The first two terms measure how well the resulted slack sizes fit to a future application (the first criterion), while the second two terms reflect the distribution of slacks (the second criterion). In order to obtain a balanced solution that favors a good fitting both on the processors and on the bus, we have used the squares of the metrics.

We call a *valid solution* that mapping and scheduling which satisfies all the design constraints (in our case the deadlines) and meets the second criterion ($C_2^\tau \geq t_{need}$ and $C_2^m \geq b_{need}$).[12]

At this point, we can give an exact formulation of our problem. Given an existing set of applications Ψ which are already mapped and scheduled, and an application $\mathcal{A}_{current}$ to be implemented on top of Ψ, we are interested in finding the subset $\Omega \subseteq \Psi$ of old applications to be remapped and rescheduled such that we produce a valid solution for $\mathcal{A}_{current} \cup \Omega$ and the total cost of modification $R(\Omega)$ is minimized. Once such a set Ω of applications is found, we are interested in optimizing the implementation of $\mathcal{A}_{current} \cup \Omega$ such that the objective function C is minimized, considering a family of future applications characterized by the sets S_t and S_b, the functions f_{S_t} and f_{S_b} as well as the parameters T_{min}, t_{need}, and b_{need}.

A mapping and scheduling strategy based on this problem formulation is presented in the following section.

15.5.5 Mapping and Scheduling Strategy

As shown in the algorithm in Figure 15.22, our mapping and scheduling strategy (MS) consists of two steps. In the first step we try to obtain a valid solution for the mapping and scheduling of $\mathcal{A}_{current} \cup \Omega$ so that the modification cost $R(\Omega)$ is minimized. Starting from such a solution, the second step iteratively improves the design in order to minimize the objective function C. In the context in which the second criterion is satisfied after the first step, improving the cost function during the second step aims at minimizing the value of $w_1^\tau(C_1^\tau)^2 + w_1^m(C_1^m)^2$.

If the first step has not succeeded in finding a solution such that the imposed time constraints are satisfied, this means that there are not sufficient resources available to implement the application $\mathcal{A}_{current}$. Thus, modifications of the system architecture have to be performed before restarting the mapping and scheduling procedure. If, however, the timing constraints are met but the second design criterion is not satisfied, a larger T_{min} (smallest expected period of a future application, see Section 15.5.3.2) or smaller values for t_{need} and/or b_{need} are suggested to the designer. This, of course, reduces the frequency of possible future applications and the amount of processor and bus resources available to them.

In the following section, we briefly discuss the basic mapping and scheduling algorithm we have used in order to generate an initial solution. The heuristic used to

[12] This definition of a valid solution can be relaxed by imposing only the satisfaction of deadlines. In this case, the algorithm in Figure 15.22 will look after a solution which satisfies the deadlines and $R(\Omega)$ is minimized; the additional second criterion is, in this case, only considered optionally.

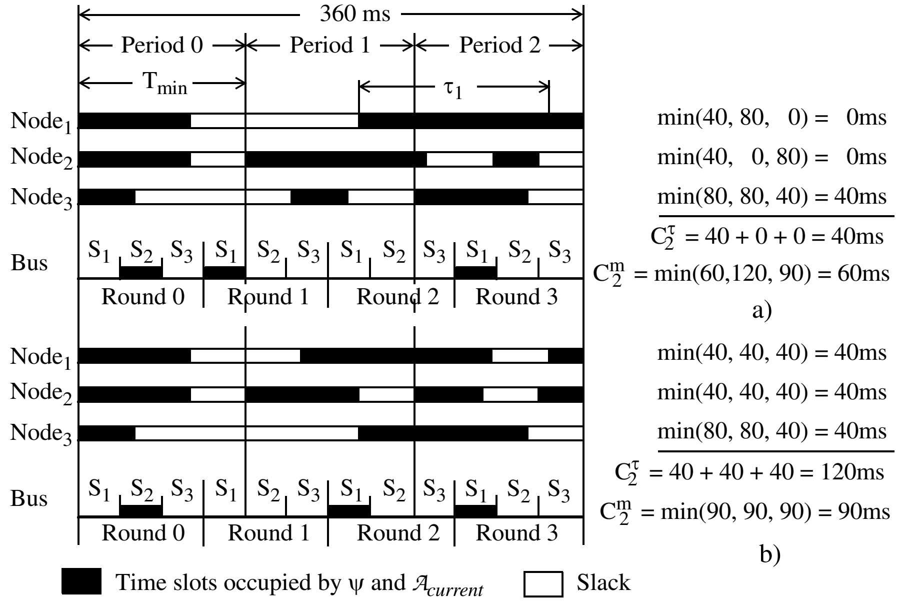


FIGURE 15.21
Example for the Second Design Criterion

iteratively improve the design with regard to the first and the second design criteria is presented in Section 15.5.5.2. In Section 15.5.5.3, we describe three alternative heuristics which can be used during the first step in order to find the optimal subset of applications to be modified.

15.5.5.1 The Initial Mapping and Scheduling

As shown in Figure 15.23, the first step of MS consists of an iteration that tries different subsets $\Omega \subseteq \Psi$ with the intention to find that subset $\Omega = \Omega_{min}$ of old applications to be remapped and rescheduled which produces a valid solution for $\mathcal{A}_{current} \cup \Omega$ such that $R(\Omega)$ is minimized. Given a subset Ω, the `InitialMappingScheduling` function (IMS) constructs a mapping and a schedule for the applications $\mathcal{A}_{current} \cup \Omega$ on top of $\Psi \backslash \Omega$, that meets the deadlines, without worrying about the two criteria introduced in Section 15.5.4.

The IMS is a classical mapping and scheduling algorithm for which we have used the Heterogeneous Critical Path (HCP) algorithm [35] as a starting point. HCP is based on a list scheduling approach [62]. We have modified the HCP algorithm in three main regards:

1. We consider that mapping and scheduling does not start with an empty system but a system on which a certain number of tasks are already mapped.

```
MappingSchedulingStrategy
  Step 1: try to find a valid solution that minimizes $R(\Omega)$
    Find a mapping and scheduling of $\mathcal{A}_{current} \cup \Omega$ on top of $\psi \setminus \Omega$ so that:
    1. constraints are satisfied;
    2. modification cost $R(\Omega)$ is minimized;
    3. the second criterion is satisfied: $C_2^\tau \geq t_{need}$ and $C_2^m \geq b_{need}$
  if Step1 has not succeeded then
    if constraints are not satisfied then
      change architecture
    else
      suggest new $T_{min}$, $t_{need}$ or $b_{need}$
    end if
    go to Step 1
  end if
  Step 2: improve the solution by minimizing objective function $C$
    Perform iteratively transformations which improve the first criterion
    (the metrics $C_1^\tau$ and $C_1^m$) without invalidating the second criterion.
end MappingSchedulingStrategy
```

FIGURE 15.22
Mapping and Scheduling Strategy (MS)

2. Messages are scheduled into bus-slots according to the TDMA protocol. The TDMA-based message scheduling technique has been presented by us in [86].

3. As a priority function for list scheduling we use, instead of the CP (critical path) priority function employed in [35], the MPCP (modified partial critical path) function introduced by us in [86]. MPCP takes into consideration the particularities of the communication protocol for calculation of communication delays.

For the example in Figure 15.17, our initial mapping and scheduling algorithm will be able to produce the optimal solution with a schedule length of 48 ms.

However, before performing the effective mapping and scheduling with IMS, two aspects have to be addressed. First, the task graphs $\mathcal{G}_i \in \mathcal{A}_{current} \cup \Omega$ have to be merged into a single graph $\mathcal{G}_{current}$ by unrolling task graphs and inserting dummy nodes as discussed in [261].

15.5.5.2 Iterative Design Transformations

Once IMS has produced a mapping and scheduling which satisfies the timing constraints, the next goal of $Step1$ is to improve the design in order to satisfy the second design criterion ($C_2^\tau \geq t_{need}$ and $C_2^m \geq b_{need}$). During the second step, the design is then further transformed with the goal of minimizing the value of $w_1^\tau(C_1^\tau)^2 + w_1^m(C_1^m)^2$, according to the requirements of the first criterion, without invalidating the second criterion achieved in the first step. In both steps, we iteratively improve the design using a transformational approach. These successive transformations are performed inside the (innermost) `repeat` loops of the first and second step, respectively (Figure 15.23). A new design is obtained from the current one by performing a transformation called *move*. We consider the following two categories of moves:

1. Moving a task to a different slack found on the same node or on a different node

2. Moving a message to a different slack on the bus

In order to eliminate those moves that will lead to an infeasible design (that violates deadlines), we do as follows. For each task τ_i, we calculate the $ASAP(\tau_i)$ and $ALAP(\tau_i)$ times considering the resources of the given hardware architecture. $ASAP(\tau_i)$ is the earliest time τ_i can start its execution, while $ALAP(\tau_i)$ is the latest time τ_i can start its execution without causing the application to miss its deadline. When moving τ_i we will consider slacks on the target processor only inside the interval $[ASAP(\tau_i), ALAP(\tau_i)]$. The same reasoning holds for messages, with the addition that a message can only be moved to slacks belonging to a slot that corresponds to the sender node (see Section 15.5.1.1). Any violation of the data dependency constraints caused by a move is rectified by shifting tasks or messages concerned in an appropriate way. If such a shift produces a deadline violation, the move is rejected.

At each step, our heuristic tries to find those moves that have the highest potential to improve the design. For each iteration, a set of potential moves is selected by the `PotentialMoveX` functions. `SelectMoveX` then evaluates these moves with regard to the respective metrics and selects the best one to be performed. We now briefly discuss the four `PotentialMoveX` functions with the corresponding moves.

PotentialMove$\mathbf{C}_2^P$ and PotentialMove$\mathbf{C}_2^m$. Consider Figure 15.21a. In $Period$ 2 on $Node_1$ there is no available slack. However, if we move task τ_1 with 40 ms to the left into $Period$ 1, as depicted in Figure 15.21b, we create a slack in $Period2$ and the periodic slack on node N_1 will be $min(40, 40, 40) = 40\ ms$, instead of 0 ms.

Potential moves aimed at improving the metric C_2^τ will be the shifting of tasks inside their $[ASAP, ALAP]$ interval in order to improve the periodic slack. The move can be performed on the same node or on less loaded nodes. The same is true for moving messages in order to improve the metric C_2^m. For the improvement of the periodic bandwidth on the bus, we also consider movement of tasks, trying to place the sender and receiver of a message on the same processor and, thus, reducing the bus load.

PotentialMove$\mathbf{C}_1^P$ and PotentialMove$\mathbf{C}_1^m$. The moves suggested by these two functions aim at improving the C_1 metric through reducing the slack fragmentation. The heuristic is to evaluate only those moves that iteratively eliminate the smallest slack in the schedule. Let us consider the example in Figure 15.24, where we have three applications mapped on a single processor: Ψ, consisting of τ_1 and τ_2, $\mathcal{A}_{current}$, having tasks τ_3, τ_4 and τ_5, and $\mathcal{A}_{future}$, with τ_6, τ_7 and τ_8. Figure 15.24 presents three possible schedules; tasks are depicted with rectangles, the width of a rectangle represents the worst-case execution time of that task. The

```
Step 1: try to find a valid solution that minimizes $R(\Omega)$
  $\Omega = \emptyset$
  repeat
    succeeded = InitialMappingScheduling($\psi \setminus \Omega$, $\mathcal{A}_{current} \cup \Omega$)
    -- compute ASAP-ALAP intervals for all tasks
    ASAP($\mathcal{A}_{current} \cup \Omega$); ALAP($\mathcal{A}_{current} \cup \Omega$)
    if succeeded then-- if time constraints are satisfied
      -- design transformations in order to satisfy
      -- the second design criterion
      repeat
        -- find set of moves with the highest potential to
        -- maximize $C_2^\tau$ or $C_2^m$
        move_set = PotentialMoveC$_2^\tau$ ($\mathcal{A}_{current} \cup \Omega$) $\cup$
          PotentialMoveC$_2^m$($\mathcal{A}_{current} \cup \Omega$)
        -- select and perform move which improves most $C_2$
        move = SelectMoveC$_2$(move_set); Perform(move)
        succeeded =$C_2^\tau \geq t_{need}$ and $C_2^m \geq b_{need}$
      until succeeded or maximum number of iterations reached
    end if
    if succeeded and $R(\Omega)$ smallest so far then
      $\Omega_{valid} = \Omega$; $solution_{valid} = solution_{current}$
    end if
        $\Omega$=NextSubset($\Omega$) -- try another subset
  until termination condition

Step 2: improve the solution by minimizing objective function $C$
  $solution_{current} = solution_{valid}$; $\Omega_{min} = \Omega_{valid}$
  -- design transformations in order to satisfy the first design criterion
  repeat
    -- find set of moves with highest potential to minimize $C_1^\tau$ or $C_1^m$
    move_set = PotentialMoveC$_1^\tau$ ($\mathcal{A}_{current} \cup \Omega_{\min}$) $\cup$
      PotentialMoveC$_2^m$($\mathcal{A}_{current} \cup \Omega_{\min}$)
    -- select move which improve $w_1^\tau(C_1^\tau)^2 + w_1^m(C_1^m)^2$,
    -- and does not invalidate the second criterion
    move = SelectMoveC$_1$(move_set); Perform(move)
  until $w_1^\tau(C_1^\tau)^2 + w_1^m(C_1^m)^2$ has not changed or
    maximum number of iterations reached
```

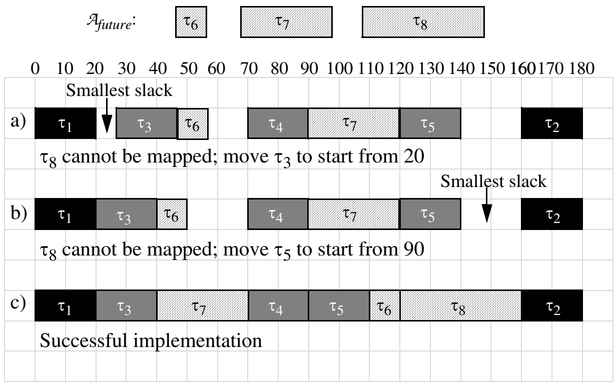
FIGURE 15.23
Step 1 and Step 2 of the Mapping and Scheduling Strategy in Figure 15.22

`PotentialMoveC`$_1$ functions start by identifying the smallest slack in the schedule table. In Figure 15.24a, the smallest slack is the slack between τ_1 and τ_3. Once the smallest slack has been identified, potential moves are investigated which either remove or enlarge the slack. For example, the slack between τ_1 and τ_3 can be removed by attaching τ_3 to τ_1, and it can be enlarged by moving τ_3 to the right in the schedule table. Moves that remove the slack are considered only if they do not lead to an invalidation of the second design criterion, measured by the C_2 metric improved in the previous step (see Figure 15.23, Step 1). Also, the slack can be enlarged only if it does not create, as a result, other unusable slack. A slack is unusable if it cannot hold the smallest object of the future application, in our case τ_6. In Figure 15.24a, the slack can be removed by moving τ_3 such that it starts from time 20, immediately after τ_1, and it can be enlarged by moving τ_3 so that it starts from 30, 40, or 50 (considering an increment which here was set by us to 10, the size of τ_6, the smallest object in $\mathcal{A}_{future}$). For each move, the improvement on the C_1 metric is calculated, and that move is selected by the $SelectMoveC_1$ function to be performed, which leads to the largest improvement on C_1 (which means the smallest value). For all the previously considered moves of τ_3, we are not able to map τ_8 which represents 50% of the $\mathcal{A}_{future}$, therefore $C_1 = 50\%$. Consequently, we can perform any of the mentioned moves, and our algorithm selects the first one investigated, the move to start τ_3 from 20, thus removing the slack. As a result of this move, the new schedule table is the one in Figure 15.24b. In the next call of the $PotentialMoveC_1$ function, the slack between τ_5 and τ_2 is identified as the smallest slack. Out of the potential moves that eliminate this slack, listed in Figure 15.24 for case b, several lead to $C_1 = 0\%$, the largest improvement. $SelectMoveC_1$ selects moving τ_5 to start from 90, and thus we are able to map task τ_8 of the future application, leading to a successful implementation in Figure 15.24c.

The previous example has only illustrated movements of tasks. Similarly, in $PotentialMoveC_1^m$, we also consider moves of messages in order to improve C_1^m. However, the movement of messages is restricted by the TDMA bus access scheme, such that a message can only be moved into a slot corresponding to the node on which it is generated.

15.5.5.3 Minimizing the Total Modification Cost

The first step of our mapping and scheduling strategy, described in Figure 15.23, iterates on successive subsets Ω searching for a valid solution which also minimizes the total modification cost $R(\Omega)$. As a first attempt, the algorithm searches for a valid implementation of $\mathcal{A}_{current}$ without disturbing the existing applications ($\Omega = \emptyset$). If no valid solution is found, successive subsets Ω produced by the function `NextSubsetset` are considered, until a termination condition is met. The performance of the algorithm, in terms of runtime and quality of the solutions produced, is strongly influenced by the strategy employed for the function `NextSubset` and the termination condition. They determine how the design space is explored while testing different subsets Ω of applications. In the following, we present three alternative strategies. The first two can be considered as situated at opposite extremes: The first



a)
Smallest slack: between τ_1 and τ_3
Potential moves: τ_3 starting at 20, having C_1^τ =50% (denoted with 20/50%), 30/50%, 40/50%, 50/50%.
Selected move: τ_3 to 20, with C_1^τ= 50%.

b)
Smallest slack: between τ_5 and τ_2
Potential moves: τ_4 to 40/37.5%, 50/37.5%, 60/37.5%, 80/37.5%, 90/37.5%, 100/37.5%; τ_5 to 90/0%, 100/0%, 110/50%, 130/50%, 140/50%, 150/0%, 160/0%.
Selected move: τ_5 to 90 with C_1^τ= 0%.

FIGURE 15.24
Successive Steps with Potential Moves for Improving C_1

one is potentially very slow but produces the optimal result while the second is very fast and possibly low quality. The third alternative is a heuristic able to produce good quality results in relatively short time, as demonstrated by the experimental results presented in Section 15.5.6.

Exhaustive Search (ES). In order to find Ω_{min}, the simplest solution is to try successively all the possible subsets $\Omega \subseteq \Psi$. These subsets are generated in ascending order of the total modification cost, starting from $\emptyset$. The termination condition is fulfilled when the first valid solution is found or no new subsets are to be generated. Since the subsets are generated in ascending order, according to their cost, the subset Ω that first produces a valid solution is also the subset with the minimum modification cost.

The generation of subsets is performed according to the graph G that characterizes the existing applications (see Section 15.5.3.1). Finding the next subset Ω, starting from the current one, is achieved by a branch and bound algorithm that, in the worst case, grows exponentially in time with the number of applications. For the example in Figure 15.20, the call to `NextSubset`($\emptyset$) will generate $\Omega = \{\mathcal{A}_7\}$ which has the smallest non-zero modification cost $R(\{\mathcal{A}_7\}) = 20$. The next generated subsets, in order, together with their corresponding total modification cost are: $R(\{\mathcal{A}_3\}) = 50$, $R(\{\mathcal{A}_3, \mathcal{A}_7\}) = 70$, $R(\{\mathcal{A}_4, \mathcal{A}_7\}) = 90$ (the inclusion of $\mathcal{A}_4$ triggers the inclusion of $\mathcal{A}_7$), $R(\{\mathcal{A}_2, \mathcal{A}_3\}) = 120$, $R(\{\mathcal{A}_2, \mathcal{A}_3, \mathcal{A}_7\}) = 140$, $R(\{\mathcal{A}_3, \mathcal{A}_4, \mathcal{A}_7\}) = 140$, $R(\{\mathcal{A}_1\}) = 150$, and so on. The total number of possible subsets according to the graph G in Figure 15.20 is 16.

This approach, while finding the optimal subset Ω, requires a large amount of computation time and can be used only with a small number of applications.

Greedy Heuristic (GH). If the number of applications is larger, a possible solution could be based on a simple greedy heuristic which, starting from $\Omega = \emptyset$, progressively enlarges the subset until a valid solution is produced. The algorithm looks at all the non-frozen applications and picks that one which, together with its dependencies, has the smallest modification cost. If the new subset does not produce a valid solution, it is enlarged by including, in the same fashion, the next application with its dependencies. This greedy expansion of the subset is continued until the set is large enough to lead to a valid solution or no application is left. For the example in Figure 15.20, the call to `NextSubset`($\emptyset$) will produce $R(\{\mathcal{A}_7\}) = 20$, and will be successively enlarged to $R(\{\mathcal{A}_7, \mathcal{A}_3\}) = 70$, $R(\{\mathcal{A}_7, \mathcal{A}_3, \mathcal{A}_2\}) = 140$ ($\mathcal{A}_4$ could have been picked as well in this step because it has the same modification cost of 70 as $\mathcal{A}_2$ and its dependence $\mathcal{A}_7$ is already in the subset), $R(\{\mathcal{A}_7, \mathcal{A}_3, \mathcal{A}_2\mathcal{A}_4\}) = 210$, and so on.

While this approach very quickly finds a valid solution, if one exists, it is possible that the resulted total modification cost is much higher than the optimal one.

Subset Selection Heuristic (SH). An intelligent selection heuristic should be able to identify the reasons due to which a valid solution has not been produced and to find the set of candidate applications which, if modified, could eliminate the problem. The failure to produce a valid solution can have two possible causes: An initial mapping which meets the deadlines has not been found, or the second criterion is not satisfied.

Let us investigate the first reason. If an application $\mathcal{A}_i$ is to meet its deadline D_i, all its tasks $\tau_j \in \mathcal{A}_i$ have to be scheduled inside their $[ASAP, ALAP]$ intervals. `InitialMappingScheduling` (IMS) fails to schedule a task inside its $[ASAP, ALAP]$ interval if there is not enough slack available on any processor, due to other tasks scheduled in the same interval. In this situation, we say that there is a *conflict* with tasks belonging to other applications. We are interested to find out which applications are responsible for conflicts encountered during the mapping and scheduling of $\mathcal{A}_{current}$, and not only that, but also which ones are *flexible* enough to be moved away in order to avoid these conflicts.

If IMS is not able to find a solution that satisfies the deadlines, it will determine a metric $\Delta_{\mathcal{A}_i}$ that characterizes both the degree of conflict and the flexibility of each application $\mathcal{A}_i \in \Psi$ in relation to $\mathcal{A}_{current}$. A set of applications Ω will be characterized, in relation to $\mathcal{A}_{current}$, by the following metric:

$$\Delta(\Omega) = \sum_{\mathcal{A}_i \in \Omega} \Delta_{\mathcal{A}_i}. \tag{15.10}$$

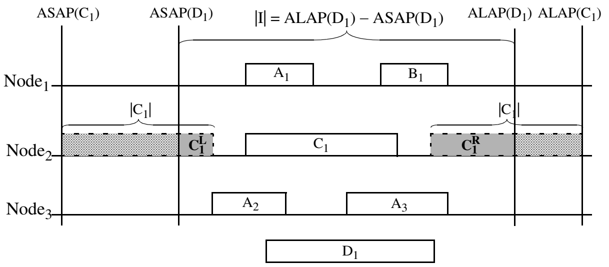
This metric $\Delta(\Omega)$ will be used by our subset selection heuristic in the case IMS has failed to produce a solution which satisfies the deadlines. An application with a larger $\Delta_{\mathcal{A}_i}$ is more likely to lead to a valid schedule if included in Ω.

In Figure 15.25, we illustrate how this metric is calculated. Applications A, B and C are implemented on a system consisting of the three processors $Node_1$, $Node_2$ and $Node_3$. The current application to be implemented is D. At a certain moment, IMS comes to the point to map and schedule task $D_1 \in D$. However, it is not able to place it inside its $[ASAP, ALAP]$ interval, denoted in Figure 15.25 as I. The reason is that there is not enough slack available inside I on any of the processors, because tasks $A_1, A_2, A_3 \in A$, $B_1 \in B$ and $C_1 \in C$ are scheduled inside that interval. We are interested to determine which of the applications A, B and C are more likely to lend free slack for D_1, if remapped and rescheduled. Therefore, we calculate the slack resulted after we move away tasks belonging to these applications from the interval I. For example, the resulted slack available after modifying application C (moving C_1 either to the left or to the right inside its own $[ASAP, ALAP]$ interval) is of size $|I| - \min(|C_1^L|, |C_1^R|)$. With $C_1^L (C_1^R)$ we denote that slice of task C_1 which remains inside the interval I after C_1 has been moved to the extreme left (right) inside its own $[ASAP, ALAP]$ interval. $|C_1^L|$ represents the length of slice C_1^L. Thus, when considering task D_1, Δ_C will be incremented with $\delta_C^{D_1} = \max(|I| - \min(|C_1^L|, |C_1^R|) - |D_1|, 0)$. This value shows the maximum theoretical slack usable for D_1, that can be produced by modifying application C. By relating this slack to the length of D_1, the value $\delta_C^{D_1}$ also captures the amount of flexibility provided by that modification.

The increments $\delta_B^{D_1}$ and $\delta_A^{D_1}$ to be added to the values of Δ_B and Δ_A, respectively, are also presented in Figure 15.25. IMS then continues the evaluation of the metrics Δ with the other tasks belonging to the current application D (with the assumption that task D_1 has been scheduled at the beginning of interval I). Thus, as a result of the failed attempt to map and schedule application D, the metrics Δ_A, Δ_B and Δ_C will be produced.

If the initial mapping was successful, the first step of MS could fail during the attempt to satisfy the second criterion (Figure 15.23). In this case, the metric $\Delta_{\mathcal{A}_i}$ is computed in a different way. What $\Delta_{\mathcal{A}_i}$ will capture in this case is the potential of an application $\mathcal{A}_i$ to improve the metric C_2 if remapped together with $\mathcal{A}_{current}$. Therefore, we consider a total number of moves from all the non-frozen applications. These moves are determined using the PotentialMoveC$_2$ functions presented in Section 15.5.5.2. Each such move will lead to a different mapping and schedule, and thus to a different C_2 value. Let us consider δ_{move} as the improvement on C_2 produced by the currently considered move. If there is no improvement, $\delta_{move} = 0$. Thus, for each move that has as subject τ_j or $m_j \in \mathcal{A}_i$, we increment the metric $\Delta_{\mathcal{A}_i}$ with the δ_{move} improvement on C_2.

As shown in the algorithm in Figure 15.23, MS starts by trying an implementation of $\mathcal{A}_{current}$ with $\Omega = \emptyset$. If this attempt fails for one of the two reasons mentioned above, the corresponding metrics $\Delta_{\mathcal{A}_i}$ are computed for all $\mathcal{A}_i \in \Psi$. Our heuristic SH will then start by finding the solution Ω_{GH} produced with the greedy heuristic GH (this will succeed if there exists any solution). The total modification cost corresponding to this solution is $R_{GH} = R(\Omega_{GH})$ and the value of the metric Δ is $\Delta_{GH} = \Delta(\Omega_{GH})$. SH now continues by trying to find a solution with a more favorable Ω than Ω_{GH} (a smaller total cost R). Therefore, the thresholds $R_{max} = R_{GH}$ and $\Delta_{min} = \Delta_{GH}/n$ (for our experiments we considered $n = 2$) are set. Sets of applications not fulfilling these thresholds will not be investigated by MS. For generating new subsets Ω, the function NextSubset now follows a similar approach



$$\delta_B^{D_1} = \max(|I| - |A_1| - \min(|B_1^L|, |B_1^R|) - |D_1|, 0); \delta_C^{D_1} = \max(|I| - \min(|C_1^L|, |C_1^R|) - |D_1|, 0)$$

$$\delta_A^{D_1} = \max(\max(\overbrace{|I| - |B_1| - \min(|A_1^L|, |A_1^R|)}^{D_1 \text{ mapped on } Node_1}), \overbrace{|I| - \min(|A_2^L|, |A_2^R|) - \min(|A_3^L|, |A_3^R|)}^{D_1 \text{ mapped on } Node_3}) - |D_1|, 0)$$

FIGURE 15.25
Metric for the Subset Selection Heuristic

like in the exhaustive search approach ES, but in a reverse direction, toward smaller subsets (starting with the set containing all non-frozen applications), and it will consider only subsets with a smaller total cost then R_{max} and a larger Δ than Δ_{min} (a small Δ means a reduced potential to eliminate the cause of the initial failure). Each time a valid solution is found, the current values of R_{max} and Δ_{min} are updated in order to further restrict the search space. The heuristic stops when no subset can be found with $\Delta > \Delta_{min}$ or a certain imposed limit has been reached (e.g., on the total number of attempts to find new subsets).

15.5.6 Experimental Results

In the following three sections, we show a series of experiments that demonstrate the effectiveness of the proposed approach and algorithms. The first set of results is related to the efficiency of our mapping and scheduling algorithm and the iterative design transformations proposed in Section 15.5.5.1 and 15.5.5.2. The second set of experiments evaluates our heuristics for minimization of the total modification cost presented in Section 15.5.5.3. As a general strategy, we have evaluated our algorithms performing experiments on a large number of test cases generated for experimental purposes. Finally, we have validated the proposed approach using a real-life example. All experiments were run on a SUN Ultra 10 workstation.

15.5.6.1 Evaluation of the IMS Algorithm and the Iterative Design Transformations

For evaluation of our approach, we used task graphs of 80, 160, 240, 320 and 400 tasks, representing the application $\mathcal{A}_{current}$, randomly generated for experimental purposes. Thirty graphs were generated for each graph dimension; thus, a total of 150 graphs were used for experimental evaluation.

We generated both graphs with random structure and graphs based on more regu-

TABLE 15.1
Evaluation of the initial mapping and scheduling.

Tasks	HCP			HCP		
	avg.	max.	better	avg.	max.	better
80	2.04%	31.57%	10%	0.35%	1.47%	30%
160	3.12%	48.89%	10%	1.18%	5.44%	33.33%
240	5.53%	61.27%	13.33%	1.38%	14.52%	36.66%
320	6.12%	88.57%	16.66%	2.79%	24.33%	40%
400	11.02%	120.77%	13.33%	2.78%	22.52%	36.66%

lar structures like trees and groups of chains. We generated a random structure graph deciding for each pair of two tasks if they should be connected or not. Two tasks in the graph were connected with a certain probability (between 0.05 and 0.15, depending on the graph dimension) on the condition that the dependency would not introduce a loop in the graph. The width of the tree-like structures was controlled by the maximum number of direct successors a task can have in the tree (from 2 to 6), while the graphs consisting of groups of chains had 2 to 12 parallel chains of tasks. Furthermore, the regular structures were modified by adding a number of 3 to 30 random cross-connections.

Execution times and message lengths were assigned randomly using both uniform and exponential distribution within the 10 to 100 ms, and 2 to 8 bytes ranges, respectively.

We considered an architecture consisting of 10 nodes of different speeds. For the communication channel, we considered a transmission speed of 256 kbps and a length below 20 meters. The maximum length of the data field in a bus slot was 8 bytes. Throughout the experiments presented in this section, we have considered an existing set of applications Ψ consisting of 400 tasks, with a schedule table of 6s on each processor, and a slack of about 50% of the total schedule size. The mapping of the existing applications has been done using a simple heuristic that tries to balance the utilization of processors while minimizing communication. The scheduling of the applications Ψ has been performed using list scheduling, and the schedules obtained have then been stretched to their deadline by introducing slacks distributed uniformly over the schedule table.

In this section, we have also considered that no modifications of the existing set of applications Ψ are allowed when implementing a new application. We will concentrate on the aspects related to the modification of existing applications in the following section.

The first result concerns the quality of the designs produced by our initial mapping and scheduling algorithm IMS. As discussed in Section 15.5.5.1, IMS uses the MPCP priority function which considers particularities of the TDMA protocol. In our experiments, we compared the quality of designs (in terms of schedule length) produced by IMS with those generated with the original HCP algorithm proposed in [35]. Results are depicted in Table 15.1 where we have three columns for both

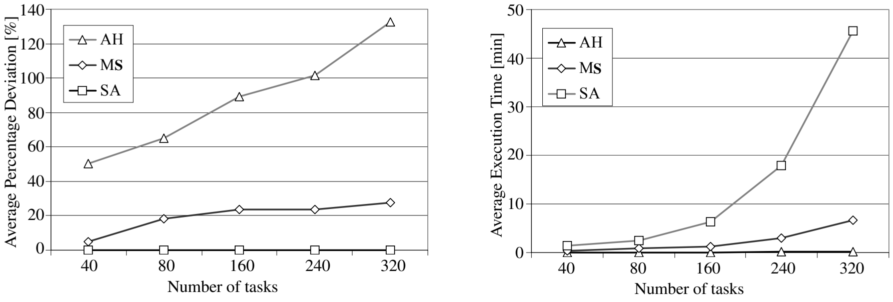
HCP and IMS. In the columns labelled "average," we present the average percentage deviations of the schedule length produced with HCP and IMS from the length of the best schedule among the two. In the maximum column, we have the maximum percentage deviation, and the column with the heading better shows the percentage of cases in which HCP or IMS was better than the other. For example, for 240 tasks, HCP had an average percentage deviation from the best result of 5.53%, compared to 1.38% for IMS. Also, in the worst case, the schedule length obtained with HCP was 61.27% larger than the one obtained with IMS. There were four cases (13.33%) in which HCP has obtained a better result than IMS, compared to 11 cases (36.66%) where IMS has obtained a better result. For the rest of the 15 cases, the schedule lengths obtained were equal. We can observe that, in average, the deviation from the best result is 3.28 times smaller with IMS than with HCP. The average execution times for both algorithms are under half a second for graphs with 400 tasks.

For the next set of experiments, we were interested to investigate the quality of the design transformation heuristic discussed in Section 15.5.5.2, aiming at the optimization of the objective function C. In order to compare this heuristic, implemented in our mapping and scheduling approach MS, we have developed two additional heuristics:

1. A *simulated annealing strategy* (SA) [275], based on the same moves as described in Section 15.5.5.2. SA is applied on the solution produced by IMS and aims at finding the near-optimal mapping and schedule that minimizes the objective function C. The main drawback of the SA strategy is that in order to find the near-optimal solution it needs very large computation times. Such a strategy, although useful for the final stages of the system synthesis, cannot be used inside a design space exploration cycle.

2. A so-called *ad-hoc approach* (AH), which is a simple, straightforward solution to produce designs that, to a certain degree, support an incremental process. Starting from the initial valid schedule of length S obtained by IMS for a graph G with N tasks, AH uses a simple scheme to redistribute the tasks inside the $[0, D]$ interval, where D is the deadline of task graph G. AH starts by considering the first task in topological order, let it be τ_1. It introduces after τ_1 a slack of size $max(smallest\ task\ size\ of\ \mathcal{A}_{future}, (D-S)/N)$, thus shifting all descendants of τ_1 to the right (toward the end of the schedule table). The insertion of slacks is repeated for the next task, with the current, larger value of S, as long as the resulted schedule has a length $S \leq D$. Processes are moved only as long as their individual deadlines (if any) are not violated.

Our heuristic (MS), as well as SA and AH have been used to map and schedule each of the 150 task graphs on the target system. For each of the resulted designs, the objective function C has been computed. Very long and expensive runs have been performed with the SA algorithm for each graph and the best ever solution produced has been considered as the near-optimum for that graph. We have compared the objective function obtained for the 150 task graphs considering each of the three heuristics. Figure 15.26a presents the average percentage deviation of the objective



a) Deviation of the objective function obtained with MS and AH from that obtained with SA

b) Execution times

FIGURE 15.26
Evaluation of the Design Transformation Heuristics

function obtained with the MS and AH from the value of the objective function obtained with the near-optimal scheme (SA). We have excluded from the results in Figure 15.26a, 37 solutions obtained with AH for which the second design criterion has not been met, and thus the objective function has been strongly penalized. The average run-times of the algorithms are presented in Figure 15.26b. The SA approach performs best in terms of quality at the expense of a large execution time: The execution time can be up to 45 minutes for large graphs of 400 tasks. The important aspect is that MS performs very well, and is able to obtain good quality solutions, very close to those produced with SA, in a very short time. AH is, of course, very fast, but since it does not address explicitly the two design criteria presented in Section 15.5.4, it has the worst quality of solutions, as expressed by the objective function.

The most important aspect of the experiments is determining to which extent the design transformations proposed by us, and the related heuristic, really facilitate the implementation of future applications. To find this out, we have mapped graphs of 80, 160, 240 and 320 nodes representing the $\mathcal{A}_{current}$ application on top of Ψ (the same Ψ as defined for the previous set of experiments). After mapping and scheduling each of these graphs, we have tried to add a new application $\mathcal{A}_{future}$ to the resulted system. $\mathcal{A}_{future}$ consists of a task graph of 80 tasks, randomly generated according to the following specifications: $S_t = \{20, 50, 100, 150, 200\ ms\}$, $f_t(S_t) = \{10, 25, 45, 15, 5\%\}$, $S_b = \{2, 4, 6, 8\ bytes\}$, $f_b(S_b) = \{20, 50, 20, 10\%\}$, $T_{min} = 250\ ms$, $t_{need} = 100$ and $b_{need} = 20\ ms$. The experiments have been performed three times: Using MS, SA and AH for mapping $\mathcal{A}_{current}$. In all three cases, we were interested to see if it is possible to find a correct implementation for $\mathcal{A}_{future}$ on top of $\mathcal{A}_{current}$ using the initial mapping and scheduling algorithm IMS (without any modification of Ψ or $\mathcal{A}_{current}$). Figure 15.27 shows the percentage of successful implementations of $\mathcal{A}_{future}$ for each of the three cases. In the case $\mathcal{A}_{current}$ has been implemented with MS and SA (this means using the design criteria and metrics proposed in the section) we were able to find a valid schedule for 65% and 68% of the total cases, respectively. However, using AH to map $\mathcal{A}_{current}$ has led to a situ-

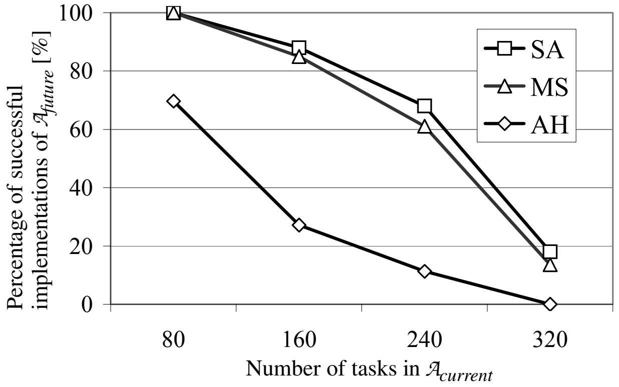


FIGURE 15.27
Percentage of Future Applications Successfully Implemented

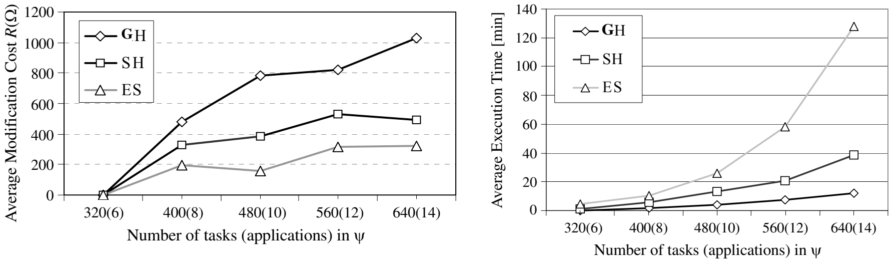
ation where IMS is able to find correct solutions in only 21% of the cases. Another conclusion from Figure 15.27 is that when the total slack available is large, as when $\mathcal{A}_{current}$ has only 80 tasks, it is easy for MS and, to a certain extent, even for AH to find a mapping that allows adding future applications. However, as $\mathcal{A}_{current}$ grows to 240 tasks, only MS and SA are able to find an implementation of $\mathcal{A}_{current}$ that supports an incremental design task, accommodating the future application in more than 60% of the cases. If the remaining slack is very small, after we map an $\mathcal{A}_{current}$ of 320 tasks, it becomes practically impossible to map new applications without modifying the current system. Moreover, our mapping heuristic MH performs very well compared to the simulated annealing approach SA which aims for the near-optimal value of the objective function.

15.5.6.2 Evaluation of the Modification Cost Minimization Heuristics

For this set of experiments, we first used the same 150 task graphs as in the previous section, consisting of 80, 160, 240, 320 and 400 tasks, for the application $\mathcal{A}_{current}$. We also considered the same system architecture as presented there.

The first results concern the quality of the solution obtained with our mapping strategy MS using the search heuristic SH compared to the case when the simple greedy approach GH and the exhaustive search ES are used. For the existing applications, we have generated five different sets Ψ, consisting of different numbers of applications and tasks, as follows: 6 applications (320 tasks), 8 applications (400 tasks), 10 applications (480 tasks), 12 applications (560 tasks), 14 applications (640 tasks). The task graphs in the applications as well as their mapping and scheduling were generated as described in the introduction to Section 15.5.6.1.

After generating the applications, we have manually assigned modification costs in the range 10 to 100, depending on their size. The dependencies between applications (in the sense introduced in Section 15.5.3.1) were such that the total



a) Modification Cost obtained with the GH, SH, and ES heuristics

b) Execution times

FIGURE 15.28
Evaluation of the Modification Cost Minimization

number of possible subsets Ω resulted for each set Ψ were 32, 128, 256, 1024 and 4096, respectively. We have considered that the future applications, $\mathcal{A}_{future}$, are characterized by the following parameters: $S_t = \{20, 50, 100, 150, 200\ ms\}$, $f_t(S_t) = \{10, 25, 45, 15, 5\%\}$, $S_b = \{2, 4, 6, 8\ bytes\}$, $f_b(S_b) = \{20, 50, 20, 10\%\}$, $T_{min} = 250\ ms$, $t_{need} = 100\ ms$ and $b_{need} = 20\ ms$.

MS has been used to produce a valid solution for each of the 150 task graphs representing $\mathcal{A}_{current}$, on each of the target configurations Ψ, using the ES, GH and SH approaches to subset selection. Figure 15.28a compares the three approaches based on the total modification cost needed in order to obtain a valid solution. The exhaustive approach ES is able to obtain valid solutions with an optimal (smallest) modification cost, while the greedy approach GH produces on average 3.12 times more costly modifications in order to obtain valid solutions. However, in order to find the optimal solution, ES needs large computation times, as shown in Figure 15.28b. For example, it can take more than two hours on average to find the smallest cost subset to be remapped that leads to a valid solution in the case of 14 applications (640 tasks). We can see that the proposed heuristic SH performs well, producing close to optimal results with a good scaling for large application sets. For the results in Figure 15.28, we have eliminated those situations in which no valid solution could be produced by MS.

Finally, we have repeated the last set of experiments discussed in the previous section (the experiments leading to the results in Figure 15.27). However, in this case, we have allowed the current system (consisting of $\Psi \cup \mathcal{A}_{current}$) to be modified when implementing $\mathcal{A}_{future}$. If the mapping and scheduling heuristic is allowed to modify the existing system, then we are able to increase the total number of successful attempts to implement application $\mathcal{A}_{future}$ from 65% to 77.5%. For the case with $\mathcal{A}_{current}$ consisting of 160 tasks (when the amount of available resources for $\mathcal{A}_{future}$ is small), the increase is from 60% to 92%. Such an increase is, of course, expected. The important aspect, however, is that it is obtained not by randomly selecting old applications to be modified, but by performing this selection such that the total modification cost is minimized.

15.6 Integration of Time-Triggered Communication with Event-Triggered Tasks

There has been a long debate in the real-time and embedded systems communities concerning the advantages of TT vs. ET approaches. Several aspects have been considered in favor of one or the other approach, such as flexibility, predictability, jitter control, processor utilization and testability. An interesting comparison of the ET and TT approaches, from a more industrial, in particular automotive, perspective, can be found in [205]. The conclusion there is that the right choice depends on the particularities of the application.

Moreover, considering preemptive priority based scheduling at the task level, with time-triggered static scheduling at the communication level, can be the right solution under certain circumstances. TT communication protocols have been classically associated with non-preemptive static scheduling of tasks, mainly for fault-tolerance reasons. A TT communication protocol, such as TTP, provides a global time-base, improves fault-tolerance and predictability. At the same time, certain particularities of the application or of the underlying real-time operating system can impose a priority based scheduling policy at the task level.

Therefore, in this section, we consider that tasks are scheduled according to a static priority preemptive policy, while messages are scheduled using a time-triggered protocol. In this section, we consider TTP-based systems, but the TT/ET integration approach is valid also for other TT protocols.

Thus, we first develop a schedulability analysis for distributed tasks with preemptive priority based scheduling considering a TTP-based communication infrastructure. Secondly, we propose four different approaches to message scheduling using static and dynamic message allocation. Finally, we show how the parameters of the communication protocol can be optimized in order to fit the communication particularities of a certain application. Thus, based on our approach, it is not only possible to determine if a certain task set implemented on a TTP-based distributed architecture is schedulable, but it is also possible to select a particular message passing strategy and also to optimize certain parameters of the communication protocol. By adapting the communication infrastructure to certain particularities of the task set, we increase the likelihood of producing an implementation which satisfies all time constraints.

15.6.1 Software Architecture

In Section 15.5.1.1, we have discussed the message passing mechanism. The organization of the message queue assembling of a frame depends on the particular approach chosen for message scheduling (see Section 15.6.3). We assume that there is a message transfer task which is activated, at certain a priori known moments, by the tick scheduler in order to perform the message transfer. Our assumption is that these activation times are stored in a message handling time table (MHTT) available to the real-time kernel in each node. Both the MEDL and the MHTT are generated off-line as a result of the schedulability analysis and optimization which will be discussed later. The MEDL imposes the times when the TTP controller of a certain node has to move frames from the MBI to the communication channel. The MHTT contains the times when messages have to be transferred by the message transfer task from the Out queue into the MBI, in order to be broadcasted by the TTP controller. As a result of this synchronization, the activation times in the MHTT are directly related to those in the MEDL and the first table results directly from the second one.

It is easy to observe that we have the most favorable situation when, at a certain activation, the message transfer task finds in the Out queue all the "expected" messages which then can be packed into the immediate following frame to be sent by the TTP controller. However, application tasks are not statically scheduled and availability of messages in the Out queue cannot be guaranteed at fixed times. Worst-case situations have to be considered, as will be shown in Section 15.6.3.

Let us consider Figure 15.16. There we assumed a context in which the broadcasting of the frame containing message m_2 is done in the slot S_0 of $Round$ 2. The TTP controller of node N_1 knows from its MEDL that it has to read a frame from slot S_0 of $Round$ 2 and to transfer it into its MBI. In order to synchronize with the TTP controller and to read the frame from the MBI, the tick scheduler on node N_1 will activate, based on its local MHTT, a so-called delivery task D. The delivery task takes the frame from the MBI and extracts the messages from it. For the case when a message is split into several packets, sent over several TDMA rounds, we consider that a message has arrived at the destination node after all its constituent packets have arrived. When m_2 has arrived, the delivery task copies it to task τ_3 which will be activated. Activation times for the delivery task are fixed in the MHTT just as explained earlier for the message transfer task.

The number of activations of the message transfer and delivery tasks depends on the number of frames transferred, and it is taken into account in our analysis, as also is the delay implied by the propagation on the communication bus.

15.6.2 Optimization Problem

We model an application as a set of tasks. Each task τ_i is allocated to a certain processor, and has a known worst-case execution time C_i, a period T_i, a deadline D_i and a uniquely assigned priority. We consider a preemptive execution environment, which means that higher priority tasks can interrupt the execution of lower priority tasks. A lower priority task can block a higher priority task (e.g., it is in its critical section), and the blocking time is computed according to the priority ceiling protocol. Tasks exchange messages, and for each message m_i we know its size S_{m_i}. A message is sent once in every n_m invocations of the sending task, with a period $T_m = n_m T_i$ inherited from the sender task τ_i, and has a unique destination task. Each task is allocated to a node of the distributed system and messages are transmitted according to the TTP.

We are interested to synthesize the MEDL of the TTP controllers (and, as a direct consequence, also the MHTTs) so that the task set is schedulable on an as cheap (slow) as possible processor set.

The next section presents the schedulability analysis for each of the four approaches considered for message scheduling, under the assumptions outlined above. In Section 15.6.4, the response times calculated using this schedulability analysis are combined in a cost function that measures the "degree of schedulability" of a given design alternative. This "degree of schedulability" is then used to drive the optimization and synthesis of the MEDL and the MHTTs.

15.6.3 Schedulability Analysis

Under the assumptions presented in the previous section, [330] integrate processor and communication scheduling and provide a "holistic" schedulability analysis in the context of distributed real-time systems with communication based on a simple TDMA protocol. The validity of this analysis has been later confirmed in [244]. The analysis belongs to the class of response time analyses, where the schedulability test is whether the worst-case response time of each task is smaller than or equal to its deadline. In the case of a distributed system, this response time also depends on the communication delay due to messages. In [330] the analysis for messages is done in a similar way as for tasks: A message is seen as an unpreemptable task that is "running" on a bus.

The basic idea in [330] is that the release jitter of a destination task depends on the communication delay between sending and receiving a message. The release jitter of a task is the worst-case delay between the arrival of the task and its release (when it is placed in the run-queue for the processor). The communication delay is the worst-case time spent between sending a message and the message arriving at the destination task.

Thus, for a task $d(m)$ that receives a message m from a sender task $s(m)$, the release jitter is

$$J_{d(m)} = r_{s(m)} + a_m + r_{deliver} + T_{tick} \tag{15.11}$$

where $r_{s(m)}$ is the response time of the task sending the message, a_m (worst-case arrival time) is the worst-case time needed for message m to arrive at the communication controller of the destination node, $r_{deliver}$ is the response time of the delivery task (see Section 15.6.1) and T_{tick} is the jitter due to the operation of the tick scheduler. The communication delay for a message m (also referred to as the "response time" of message m) is

$$r_m = a_m + r_{deliver} \tag{15.12}$$

where a_m itself is the sum of the access delay Y_m and the propagation delay X_m. The access delay is the time a message queued at the sending processor spends waiting for the use of the communication channel. In a_m, we also account for the execution time of the message transfer task (see Section 15.6.1). The propagation delay is the time taken for the message to reach the destination processor once physically sent by the corresponding TTP controller. The analysis assumes that the period T_m of any message m is longer than or equal to the length of a TDMA round, $T_m \geq T_{TDMA}$ (see Figure 15.29).

The pessimism of this analysis can be reduced by using the notion of offset in order to model precedence relations between tasks [328]. The basic idea is to exclude certain scenarios which are impossible due to precedence constraints. By considering dynamic offsets, the tightness of the analysis can be further improved [117, 118]. In the present section, our attention is concentrated on the analysis of network communication delays and on optimization of message passing strategies. In order to keep the discussion focused, we present our analysis starting from the results in [330]. All the conclusions of this research apply as well to the developments addressing precedence relations proposed, for example, in [117, 118].

Although there are many similarities with the general TDMA protocol, the analysis in the case of TTP is different in several aspects and also differs to a large degree depending on the policy chosen for message scheduling.

Before going into details for each of the message scheduling approaches proposed by us, we analyze the propagation delay and the message transfer and delivery tasks, as they do not depend on the particular message scheduling policy chosen. The propagation delay X_m of a message m sent as part of a slot S, with the TTP protocol, is equal to the time needed for the slot S to be transferred on the bus (this is the slot size expressed in time units; see Figure 15.29). This time depends on the number of bits which can be packed into the slot and on the features of the underlying bus.

The overhead produced by the communication activities must be accounted not only as part of the access delay for a message, but also through its influence on the response time of tasks running on the same processor. We consider this influence during the schedulability analysis of processes on each processor. We assume that the worst-case computation time of the transfer task (T in Figure 15.16) is known, and that it is different for each of the four message scheduling approaches. Based on the respective MHTT, the transfer task is activated for each frame sent. Its worst-case period is derived from the minimum time between successive frames.

The response time of the delivery task (D in Figure 15.16), $r_{deliver}$, is part of the communication delay (Equation 15.12). The influence due to the delivery task must also be included when analyzing the response time of the tasks running on the respective processor. We consider the delivery task during the schedulability analysis in the same way as the message transfer task.

The response times of the communication and delivery tasks are calculated, as for all other tasks, using the arbitrary deadline analysis from [330].

The four approaches we propose for scheduling of messages using TTP differ in the way the messages are allocated to the communication channel (either statically or dynamically) and whether they are split or not into packets for transmission. The next subsections present the analysis for each approach as well as the degrees of liberty a designer has, in each of the cases, for optimizing the MEDL.

15.6.3.1 Static Single Message Allocation (SM)

The first approach for scheduling messages using TTP is to statically (offline) schedule each of the messages into a slot of the TDMA cycle, corresponding to the node sending the message. This means that for each message we decide offline to allocate

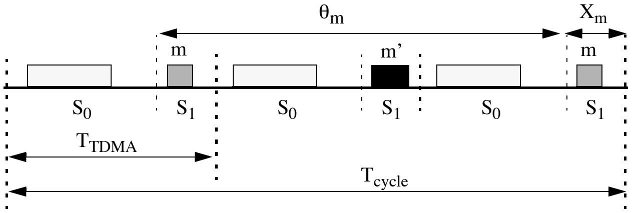


FIGURE 15.29
Worst-Case Arrival Time for SM

space in one or more frames, space that can only be used by that particular message. In Figure 15.29, the frames are denoted by rectangles. In this particular example, it has been decided to allocate space for message m in slot S_1 of the first and third rounds. Since the messages are dynamically produced by the tasks, the exact moment a certain message is generated cannot be predicted. Thus, it can happen that certain frames will be left empty during execution. For example, if there is no message m in the Out queue (see Figure 15.29) when the slot S_1 of the first round in Figure 15.29 starts, that frame will carry no information. A message m produced immediately after slot S_1 has left, could then be carried by the frame scheduled in the slot S_1 of the third round.

In the SM approach, we consider that each slot can hold a maximum of one single message. This approach is well suited for application areas, like safety-critical automotive electronics, where the messages are typically short and the ability to easily diagnose the system (fewer messages in a frame are easier to observe) is critical. In the automotive electronics area, messages are typically a couple of bytes, encoding signals like vehicle speed. However, for applications using larger messages, the SM approach leads to overheads due to the inefficient utilization of slot space when transmitting smaller size messages.

As each slot carries only one fixed, predetermined message, there is no interference among messages. If a message m misses its allocated frame, it has to wait for the following slot assigned to m. The worst-case access delay Y_m for a message m in this approach is the maximum time between consecutive slots of the same node carrying the message m. We denote this time by θ_m, illustrated in Figure 15.29, where we have a system cycle of length T_{cycle}, consisting of three TDMA rounds.

In this case, the worst-case arrival time a_m of a message m becomes $\theta_m + X_m$. Therefore, the main aspect influencing schedulability of the messages is the way they are statically allocated to slots, which determines the values of θ_m. θ_m, as well as X_m, depend on the slot sizes which in the case of SM are determined by the size of the largest message sent from the corresponding node plus the bits for control and CRC, as imposed by the protocol.

As mentioned before, the analysis in [330], done for a simple TDMA protocol, assumes that $T_m \geq T_{TDMA}$. In the case of static message allocation with TTP (the SM and MM approaches), this translates to the condition $T_m \geq \theta_m$.

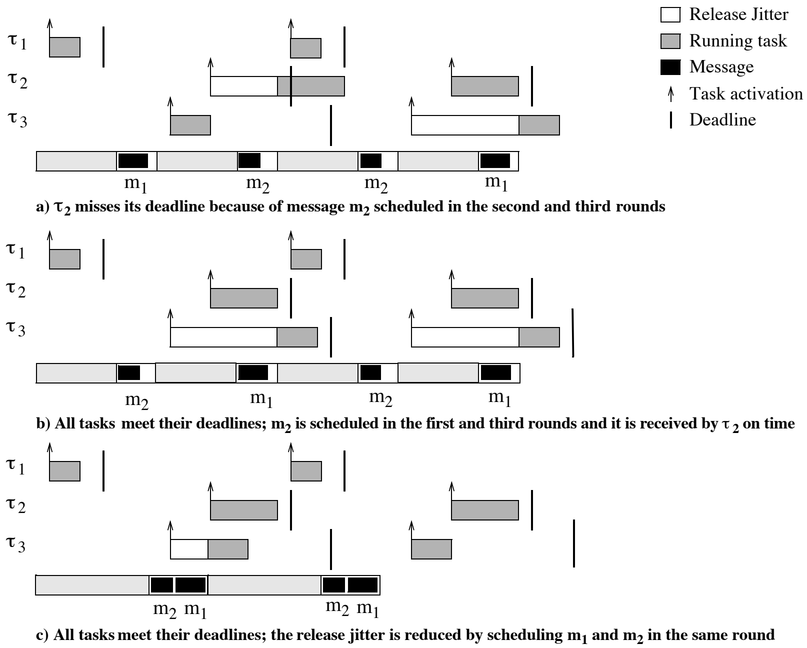

τ1
τ2
τ3
Release Jitter
Running task
Message
Task activation
Deadline
m1
m2
m2
m1
a) τ2 misses its deadline because of message m2 scheduled in the second and third rounds
m2
m1
m2
m1
b) All tasks meet their deadlines; m2 is scheduled in the first and third rounds and it is received by τ 2 on time
m2 m1
m2 m1
c) All tasks meet their deadlines; the release jitter is reduced by scheduling m1 and m2 in the same round

FIGURE 15.30
Optimizing the MEDL for SM and MM

During the synthesis of the MEDL, the designer has to allocate the messages to slots in such a way that the task set is schedulable. Since the schedulability of the task set can be influenced by the synthesis of the MEDL only through the θ_m parameters, these are the parameters which have to be optimized.

Let us consider the simple example depicted in Figure 15.30, where we have three tasks, τ_1, τ_2 and τ_3 each running on a different processor. When task τ_1 finishes executing, it sends message m_1 to task τ_3 and message m_2 to task τ_2. In the TDMA configurations presented in Figure 15.30, only the slot corresponding to the CPU running τ_1 is important for our discussion and the other slots are represented with light gray. With the configuration in Figure 15.31a, where the message m_1 is allocated to the rounds 1 and 4 and the message m_2 is allocated to rounds 2 and 3, task τ_2 misses its deadline because of the release jitter due to the message m_2 in $Round$ 2. However, if we have the TDMA configuration depicted in Figure 15.30b, where m_1 is allocated to rounds 2 and 4 and m_2 is allocated to rounds 1 and 3, all the tasks meet their deadlines.

15.6.3.2 Static Multiple Message Allocation (MM)

This second approach is an extension of the first one. In this approach, we allow more than one message to be statically assigned to a slot and all the messages transmitted in the same slot are packaged together in a frame. As for the SM approach, there is no interference among messages, so the worst-case access delay for a message m is the maximum time between consecutive slots of the same node carrying the message m, θ_m. It is also assumed that $T_m \geq \theta_m$.

However, this approach offers more freedom during the synthesis of the MEDL. We have now to decide also on how many and which messages should be put in a slot. This allows more flexibility in optimizing the θ_m parameter. To illustrate this, let us consider the same example depicted in Figure 15.31. With the MM approach, the TDMA configuration can be arranged as depicted in Figure 15.30c, where the messages m_1 and m_2 are put together in the same slot in the rounds 1 and 2. Thus, the deadline is met and the release jitter is further reduced compared to the case presented in Figure 15.31b where task τ_3 was experiencing a large release jitter.

15.6.3.3 Dynamic Message Allocation (DM)

The previous two approaches have statically allocated one or more messages to their corresponding slots. This third approach considers that the messages are dynamically allocated to frames, as they are produced.

Thus, when a message is produced by a sender task, it is placed in the Out queue (Figure 15.16). Messages are ordered according to their priority. At its activation, the message transfer task takes a certain number of messages from the head of the Out queue and constructs the frame. The number of messages accepted is decided so that their total size does not exceed the length of the data field of the frame. This length is limited by the size of the slot corresponding to the respective processor. Since the messages are sent dynamically, we have to identify them in a certain way so that they are recognized when the frame arrives at the delivery task. We consider that each message has several identifier bits appended at the beginning of the message.

Since we dynamically package messages into frames in the order they are sorted in the queue, the access delay to the communication channel for a message m depends on the number of messages queued ahead of it.

The analysis in [330] bounds the number of queued ahead *packets* of messages of higher priority than message m, as in their case it is considered that a message can be split into packets before it is transmitted on the communication channel. We use the same analysis but we have to apply it for the number of *messages* instead of packets. We have to consider that messages can be of different sizes, as opposed to packets which are always of the same size.

Therefore, the total *size* of higher priority messages queued ahead of a message m, in the worst case, is:

$$I_m = \sum_{\forall j \in hp(m)} \left\lceil \frac{r_{s(j)}}{T_j} \right\rceil S_j \qquad (15.13)$$

where S_j is the size of the message m_j, $r_{s(j)}$ is the response time of the task sending message m_j and T_j is the period of the message m_j.

Further, we calculate the worst-case time that a message m spends in the Out queue. The number of TDMA rounds needed, in the worst case, for a message m

placed in the queue to be removed from the queue for transmission is

$$\left\lceil \frac{S_m + I_m}{S_s} \right\rceil \tag{15.14}$$

where S_m is the size of the message m and S_s is the size of the slot transmitting m (we assume, in the case of DM, that for any message x, $S_x \leq S_S$). This means that the worst-case time a message m spends in the Out queue is given by

$$Y_m = \left\lceil \frac{S_m + I_m}{S_s} \right\rceil T_{TDMA} \tag{15.15}$$

where T_{TDMA} is the time taken for a TDMA round.

Since the size of the messages is given with the application, the parameter that will be optimized during the synthesis of the MEDL is the slot size. To illustrate how the slot size influences schedulability, let us consider the example in Figure 15.31 where we have the same setting as for the example in Figure 15.30. The difference is that we consider message m_1 having a higher priority than message m_2 and we schedule the messages dynamically as they are produced. With the configuration in Figure 15.31a, message m_1 will be dynamically scheduled first in the slot of the first round, while message m_2 will wait in the Out queue until the next round comes, thus causing task τ_2 to miss its deadline. However, if we enlarge the slot so that it can accommodate both messages, message m_2 does not have to wait in the queue and it is transmitted in the same slot as m_1. Therefore, τ_2 will meet its deadline as presented in Figure 15.31b. However, in general, increasing the length of slots does not necessarily improve schedulability, as it delays the communication of messages generated by other nodes.

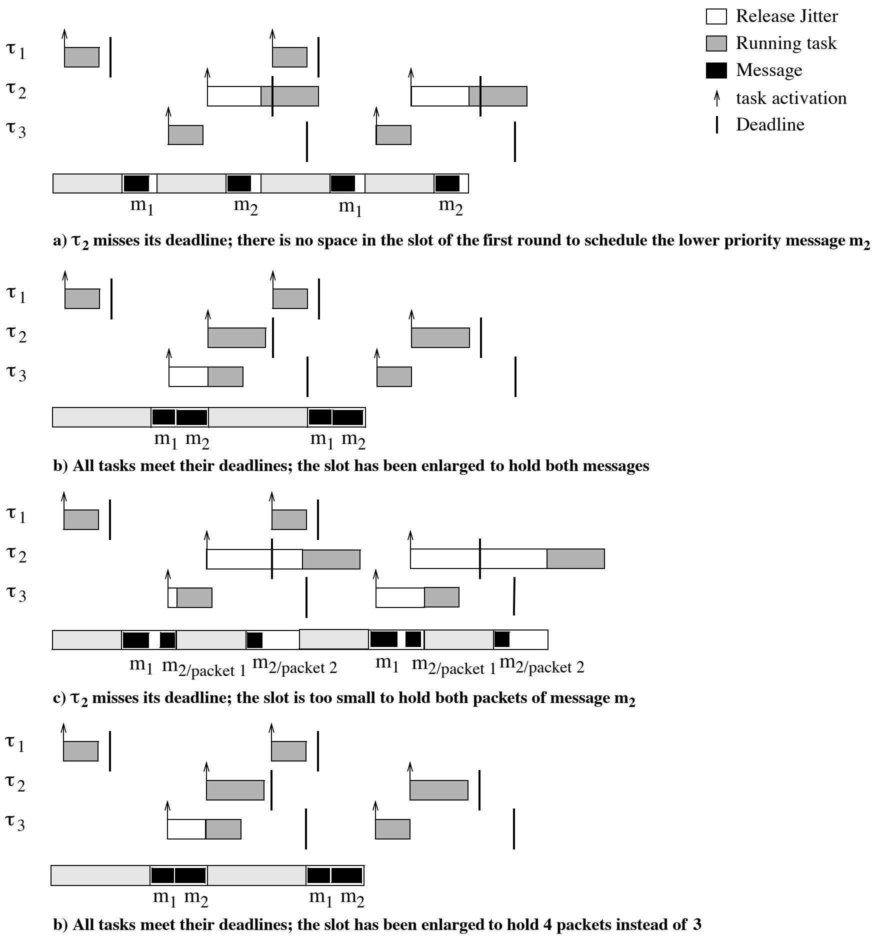
15.6.3.4 Dynamic Packet Allocation (DP)

This approach is an extension of the previous one, as we allow the messages to be split into packets before they are transmitted on the communication channel. We consider that each slot has a size that accommodates a frame with the data field being a multiple of the packet size. This approach is well suited for the application areas that typically have large message sizes. By splitting messages into packets, we can obtain a higher utilization of the bus and reduce the release jitter. However, since each packet has to be identified as belonging to a message, and messages have to be split at the sender and reconstructed at the destination, the overhead becomes higher than in the previous approaches.

The worst-case time a message m spends in the Out queue is given by the analysis in [330] which is based on similar assumptions as those for this approach:

$$Y_m = \left\lceil \frac{p_m + I_m}{S_p} \right\rceil T_{TDMA} \tag{15.16}$$

where p_m is the number of packets of message m, S_p is the size of the slot (in number



a) τ_2 misses its deadline; there is no space in the slot of the first round to schedule the lower priority message m_2

b) All tasks meet their deadlines; the slot has been enlarged to hold both messages

c) τ_2 misses its deadline; the slot is too small to hold both packets of message m_2

b) All tasks meet their deadlines; the slot has been enlarged to hold 4 packets instead of 3

FIGURE 15.31
Optimizing the MEDL for DM and DP

of packets) corresponding to m and

$$I_m = \sum_{\forall j \in hp(m)} \left\lceil \frac{r_{s(j)}}{T_j} \right\rceil p_j \tag{15.17}$$

where p_j is the number of packets of a message m_j.

In the previous approach (DM), the optimization parameter for the synthesis of the MEDL was the size of the slots. With this approach, we can also decide on the packet size which becomes another optimization parameter. Consider the example in Figure 15.31c where messages m_1 and m_2 have a size of 6 bytes each. The packet size is considered to be 4 bytes and the slot corresponding to the messages has a size of 12 bytes (3 packets) in the TDMA configuration. Since message m_1 has a higher priority than m_2, it will be dynamically scheduled first in the slot of the first round and it will need 2 packets. In the third packet, the first 4 bytes of m_2 are placed. Thus, the remaining 2 bytes of message m_2 have to wait for the next round, causing task τ_2 to miss its deadline. However, if we change the packet size to 3 bytes and keep the same size of 12 bytes for the slot, we have 4 packets in the slot corresponding to the CPU running τ_1 (Figure 15.31d). Message m_1 will be dynamically scheduled first and will need 2 packets in the slot of the first round. Hence, m_2 can be sent in the same round so that τ_2 can meet its deadline.

In this particular example, with one single sender processor and the particular message and slot sizes as given, the problem seems to be simple. This is, however, not the case in general. For example, the packet size which fits a particular node can be unsuitable in the context of the messages and slot size corresponding to another node. At the same time, reducing the packets size increases the overheads due to the transfer and delivery tasks.

The analysis presented so far is valid only in the case the arrival time a_m of a message m is smaller than or equal to its period T_m. However, in the case $a_m > T_m$ the "arbitrary deadline" analysis from [196] has to be used. We have shown in [262] how the analysis presented here can be extended to consider arbitrary deadlines.

15.6.4 Optimization Strategy

Our problem is to analyze the schedulability of a given task set and to synthesize the MEDL of the TTP controllers (and consequently the MHTTs) so that the task set is schedulable on an as cheap as possible architecture. The optimization is performed on the parameters which have been identified for each of the four approaches to message scheduling discussed before. In order to guide the optimization task, we need a cost function that captures the "degree of schedulability" for a certain MEDL implementation. Our cost function is similar to that in [329] in the case an application is not schedulable (f_1). However, in order to distinguish between several schedulable applications, we have introduced the second expression, f_2, which measures, for a feasible design alternative, the total difference between the response times and the deadlines:

$$cost(optimization\,parameters) = \begin{cases} f_1 = \sum\limits_{i=1}^{n} \max(0, R_i - D_i), \text{if } f_1 > 0 \\ f_2 = \sum\limits_{i=1}^{n} R_i - D_i), \text{if } f_1 = 0 \end{cases} \tag{15.18}$$

where n is the number of tasks in the application, R_i is the response time of a task τ_i and D_i is the deadline of a task τ_i. If the task set is not schedulable, there exists at least one R_i that is greater than the deadline D_i; therefore, the term f_1 of the function will be positive. In this case, the cost function is equal to f_1. However, if the task set is schedulable, then all R_i are smaller than the corresponding deadlines D_i. In this case, $f_1 = 0$ and we use f_2 as the cost function, as it is able to differentiate between two alternatives, both leading to a schedulable task set. For a given set of optimization parameters leading to a schedulable task set, a smaller f_2 means that we have improved the response times of the tasks, so the application can be potentially implemented on a cheaper hardware architecture (with slower processors and/or bus, but without increasing the number of processors or buses).

The response time R_i is calculated according to the arbitrary deadline analysis [330] based on the release jitter of the tasks (see Section 15.6.3), its worst-case execution time, the blocking time, and the interference time due to higher priority tasks. They form a set of mutually dependent equations which can be solved iteratively. As shown in [330], a solution can be found if the processor utilization is less than 100%.

For a given application, we are interested to synthesize a MEDL such that the cost function is minimized. We are also interested to evaluate in different contexts the four approaches to message scheduling, thus offering the designer a decision support for choosing the approach that best fits his application.

The MEDL synthesis problem belongs to the class of exponential complexity problems; therefore, we are interested to develop heuristics that are able to find accurate results in a reasonable time. We have developed optimization algorithms corresponding to each of the four approaches to message scheduling. A first set of algorithms presented in Section 15.6.4.1 is based on simple and fast greedy heuristics. In Section 15.6.4.2, we introduce a second class of heuristics which aims at finding near-optimal solutions using the simulated annealing (SA) algorithm.

15.6.4.1 Greedy Heuristics

We have developed greedy heuristics for each of the four approaches to message scheduling. The main idea of the heuristics is to minimize the cost function by incrementally trying to reduce the communication delay of messages and, by this, the release jitter of the tasks.

The only way to reduce the release jitter in the SM and MM approaches is through the optimization of the θ_m parameters. This is achieved by a proper placement of messages into slots (see Figure 15.30).

The `OptimizeSM` algorithm presented in Figure 15.32 starts by deciding on a

```
OptimizeSM
    -- set the slot sizes
    for each node $N_i$ do
        $size_{S_i}$ = max(size of messages $m_j$ sent by node $N_i$)
    end for
    -- find the min. no. of rounds that can hold all the messages
    for each node $N_i$ do
        $nm_i$ = number of messages sent from $N_i$
    end for
    $MinRounds$ = max ($nm_i$)
    -- create a minimal complete MEDL
    for each message $m_i$
        find $round$ in [1..$MinRounds$] that has an empty slot for $m_i$
        place $m_i$ into its slot in $round$
    end for
    for each $RoundsNo$ in [$MinRounds$...$MaxRounds$] do
        -- insert messages in such a way that the cost is minimized
        repeat
            for each task $P_i$ that receives a message $m_i$ do
                if $D_i$ - $R_i$ is the smallest so far then $m = m_{P_i}$ end if
            end for
            for each $round$ in [1..$RoundsNo$] do
                place $m$ into its corresponding slot in $round$
                calculate the $CostFunction$
                if the $CostFunction$ is smallest so far then
                    $BestRound$ = $round$
                end if
                remove $m$ from its slot in $round$
            end for
            place $m$ into its slot in $BestRound$ if one was identified
        until the $CostFunction$ is not improved
    end for
end OptimizeSM
```

FIGURE 15.32
Greedy Heuristic for SM

size ($size_{S_i}$) for each of the slots. The slot sizes are set to the minimum size that can accommodate the largest message sent by the corresponding node (lines 1–4 in Figure 15.32). In this approach, a slot can carry at most one message; thus, slot sizes larger than this size would lead to larger response times.

Then, the algorithm has to decide on the number of rounds, thus determining the size of the MEDL. Since the size of the MEDL is physically limited, there is a limit to the number of rounds (e.g., 2, 4, 8, 16 depending on the particular TTP controller implementation). However, there is a minimum number of rounds $MinRounds$ that is necessary for a certain application, which depends on the number of messages transmitted (lines 5–9). For example, if the tasks mapped on node N_0 send in total seven messages then we have to decide on at least seven rounds in order to accommodate all of them (in the SM approach there is at most one message per slot). Several numbers of rounds, $RoundsNo$, are tried out by the algorithm starting from $MinRounds$ up to $MaxRounds$ (lines 15–31).

For a given number of rounds (that determine the size of the MEDL), the initially empty MEDL has to be populated with messages in such a way that the cost function is minimized. In order to apply the schedulability analysis that is the basis for the cost function, a *complete* MEDL has to be provided. A complete MEDL contains at least one instance of every message that has to be transmitted between the tasks on different processors. A *minimal complete* MEDL is constructed from an empty MEDL by placing one instance of every message m_i into its corresponding empty slot of a round (lines 10–14). In Figure 15.30a, for example, we have a MEDL composed of four rounds. We get a minimal complete MEDL, for example, by assigning m_2 and m_1 to the slots in rounds 3 and 4, and leaving the slots in rounds 1 and 2 empty. However, such a MEDL might not lead to a schedulable system. The "degree of schedulability" can be improved by inserting instances of messages into the available places in the MEDL, thus minimizing the θ_m parameters. For example, in Figure 15.30a inserting another instance of the message m_1 in the first round and m_2 in the second round leads to τ_2 missing its deadline, while in Figure 15.30b inserting m_1 into the second round and m_2 into the first round leads to a schedulable system.

Our algorithm repeatedly adds a new instance of a message to the current MEDL in the hope that the cost function will be improved (lines 16–30). In order to decide an instance of which message should be added to the current MEDL, a simple heuristic is used. We identify the task τ_i which has the most "critical" situation, meaning that the difference between its deadline and response time, $D_i - R_i$, is minimal compared with all other tasks. The message to be added to the MEDL is the message $m = m_{P_i}$ received by the task τ_i (lines 18–20). Message m will be placed into that round ($BestRound$) which corresponds to the smallest value of the cost function (lines 21–28). The algorithm stops if the cost function cannot be further improved by adding more messages to the MEDL.

The `OptimizeMM` algorithm is similar to `OptimizeSM`. The main difference is that in the MM approach several messages can be placed into a slot (which also decides its size), while in the SM approach there can be at most one message per slot. Also, in the case of MM, we have to take additional care that the slots do not exceed the maximum allowed size for a slot.

```
   OptimizeDM
01   for each node $N_i$ do
02     $MinSize_{Si}$ = max(size of messages $m_j$ sent by node $N_i$)
03   end for
04   -- identifies the size that minimizes the cost function
05   for each slot $S_i$
06     $BestSize_{Si}$ = $MinSize_{Si}$
07     for each $SlotSize$ in [$MinSize_{Si}$...$MaxSize$] do
08       calculate the $CostFunction$
09       if the $CostFunction$ is best so far then
10         $BestSize_{Si}$ = $SlotSize_{Si}$
11       end if
12     end for
13     $size_{Si}$ = $BestSize_{Si}$
14   end for
   end OptimizeDM
```

FIGURE 15.33
Greedy Heuristic for DM

The situation is simpler for the dynamic approaches, namely DM and DP, since we only have to decide on the slot sizes and, in the case of DP, on the packet size. For these two approaches, the placement of messages is dynamic and has no influence on the cost function. The `OptimizeDM` algorithm (see Figure 15.33) starts with the first slot $S_i = S_0$ of the TDMA round and tries to find that size ($BestSize_{S_i}$) which corresponds to the smallest $CostFunction$ (lines 4–14 in Figure 15.33). This slot size has to be large enough ($S_i \geq MinSize_{S_i}$) to hold the largest message to be transmitted in this slot, and within bounds determined by the particular TTP controller implementation (e.g., from 2 bits up to $MaxSize = 32$ bytes). Once the size of the first slot has been determined, the algorithm continues in the same manner with the next slots (lines 7–12).

The `OptimizeDP` algorithm has also to determine the proper packet size. This is done by trying all the possible packet sizes given the particular TTP controller. For example, it can start from 2 bits and increment with the "smallest data unit" (typically 2 bits) up to 32 bytes. In the case of the `OptimizeDP` algorithm, the slot size has to be determined as a multiple of the packet size and within certain bounds depending on the TTP controller.

15.6.4.2 Simulated Annealing Strategy

We have also developed an optimization procedure based on a simulated annealing (SA) strategy. The main characteristic of such a strategy is that it tries to find the global optimum by randomly selecting a new solution from the neighbors of the current solution. The new solution is accepted if it is an improved one. However, a worse

```
SimulatedAnnealing
01  construct an initial TDMA round $x^{now}$
02  temperature = initial temperature TI
03  repeat
04      for i = 1 to temperature length TL
05          generate randomly a neighboring solution x' of $x^{now}$
06          delta = CostFunction(x') - CostFunction($x^{now}$)
07          if delta < 0 then $x^{now}$ = x'
08          else
09              generate q = random (0, 1)
10              if $q < e^{-delta / temperature}$ then $x^{now}$ = x' end if
11          end if
12      end for
13      temperature = $\alpha$ * temperature
14  until stopping criterion is met
15  return solution corresponding to the best CostFunction
end SimulatedAnnealing
```

FIGURE 15.34
The Simulated Annealing Strategy

solution can also be accepted with a certain probability that depends on the deterioration of the cost function and on a control parameter called temperature [275].

In Figure 15.34, we give a short description of this algorithm. An essential component of the algorithm is the generation of a new solution x starting from the current one x^{now} (line 5 in Figure 15.34). The neighbors of the current solution x^{now} are obtained depending on the chosen message scheduling approach. For SM, x is obtained from x^{now} by inserting or removing a message in one of its corresponding slots. In the case of MM, we have to take additional care that the slots do not exceed the maximum allowed size (which depends on the controller implementation), as we can allocate several messages to a slot. For these two static approaches, we also decide on the number of rounds in a cycle (e.g., 2, 4, 8, 16; limited by the size of the memory implementing the MEDL). In the case of DM, the neighboring solution is obtained by increasing or decreasing the slot size within the bounds allowed by the particular TTP controller implementation, while in the DP approach we also increase or decrease the packet size.

For the implementation of this algorithm, the parameters TI (initial temperature), TL (temperature length), α (cooling ratio) and the stopping criterion have to be determined. They define the so called cooling schedule and have a strong impact on the quality of the solutions and the CPU time consumed. We were interested to obtain values for TI, TL and α that will guarantee the finding of good quality solutions in a short time. In order to tune the parameters, we have first performed very long and expensive runs on selected large examples and the best ever solution, for each example, has been considered as the near-optimum. Based on further experiments, we have determined the parameters of the SA algorithm, for different sizes of examples, so that the optimization time is reduced as much as possible but the near-optimal result is still produced. These parameters have then been used for the large-scale experiments presented in the following section. For example, for the graphs with 320 nodes, TI is 300, TL is 500 and α is 0.95. The algorithm stops if for three consecutive temperatures no new solution has been accepted.

15.6.5 Experimental Results

For evaluation of our approaches, we first used sets of tasks generated for experimental purposes. We considered architectures consisting of 2, 4, 6, 8 and 10 nodes. Forty tasks were assigned to each node, resulting in sets of 80, 160, 240, 320 and 400 tasks. Thirty tasks sets were generated for each of the five dimensions. Thus, a total of 150 sets of tasks were used for experimental evaluation. Worst-case computation times, periods, deadlines and message lengths were assigned randomly within certain intervals. For the communication channel, we considered a transmission speed of 256 kbps. The maximum length of the data field in a slot was 32 bytes and the frequency of the TTP controller was chosen to be 20 MHz. All experiments were run on a Sun Ultra 10 workstation.

For each of the 150 generated examples and each of the four message scheduling approaches, we have obtained the near-optimal values for the cost function (Equation 15.18) as produced by our SA based algorithm (see Section 15.6.4.2). For a given example, these values might differ from one message passing approach to another, as they depend on the optimization parameters and the schedulability analysis which are particular for each approach. Figure 15.35 presents a comparison based on the average percentage deviation of the cost function obtained for each of the four approaches, from the minimal value among them. The percentage deviation is calculated according to the formula:

$$deviation = \frac{cost_{approach} - cost_{best}}{cost_{best}} \times 100. \tag{15.19}$$

The DP approach is, generally, able to achieve the highest degree of schedulability, which in Figure 15.35 translates in the smallest deviation. In the case the packet size is properly selected, by scheduling messages dynamically we are able to efficiently use the available space in the slots, and thus reduce the release jitter. However, by using the MM approach we can obtain almost the same result if the messages are carefully allocated to slots as does our optimization strategy.

Moreover, in the case of larger task sets, the static approaches suffer significantly less overhead than the dynamic approaches. In the SM and MM approaches, the messages are uniquely identified by their position in the MEDL. However, for the dynamic approaches we have to somehow identify the dynamically transmitted messages and packets. Thus, for the DM approach we consider that each message has several identifier bits appended at the beginning of the message, while for the DP approach the identification bits are appended to each packet. Not only do the identifier bits add to the overhead, but in the DP approach, the transfer and delivery tasks

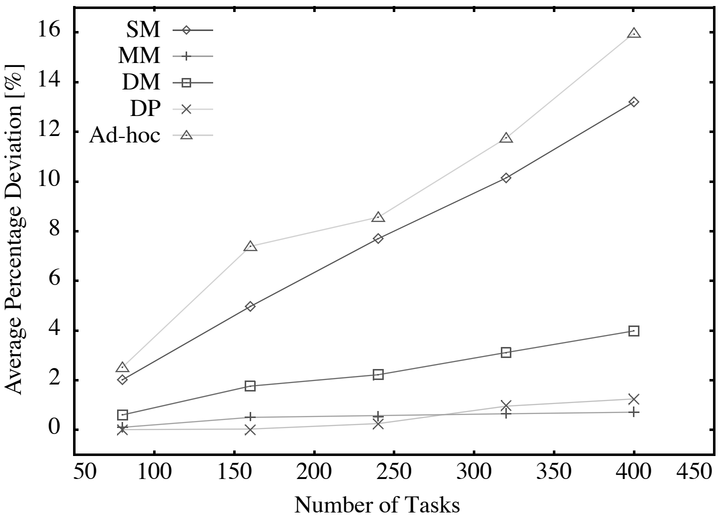


FIGURE 15.35
Comparison of the Four Approaches to Message Scheduling

(see Figure 15.16) have to be activated at each sending and receiving of a packet, and thus interfere with the other tasks. Thus, for larger applications (e.g., task sets of 400 tasks), MM outperforms DP, as DP suffers from large overhead due to its dynamic nature. DM performs worse than DP because it does not split the messages into packets, and this results in a mismatch between the size of the messages dynamically queued and the slot size, leading to unused slot space that increases the jitter. SM performs the worst as it does not permit much room for improvement, leading to large amounts of unused slot space. Also, DP has produced a MEDL that resulted in schedulable task sets for 1.33 times more cases than the MM and DM. MM, in its turn, produced two times more schedulable results than the SM approach.

Together with the four approaches to message scheduling, a so-called ad-hoc approach is presented. The ad-hoc approach performs scheduling of messages without trying to optimize the access to the communication channel. The ad-hoc solutions are based on the MM approach and consider a design with the TDMA configuration consisting of a simple, straightforward allocation of messages to slots. The lengths of the slots were selected to accommodate the largest message sent from the respective node. Figure 15.35 shows that the ad-hoc alternative is constantly outperformed by any of the optimized solutions. This demonstrates that significant gains can be obtained by optimization of the parameters defining the access to the communication channel.

Next, we have compared the four approaches with respect to the number of messages exchanged between different nodes and the maximum message size allowed. For the results depicted in Figures 15.36 and 15.37, we have assumed sets of 80 tasks allocated to four nodes. Figure 15.36 shows that, as the number of messages in-

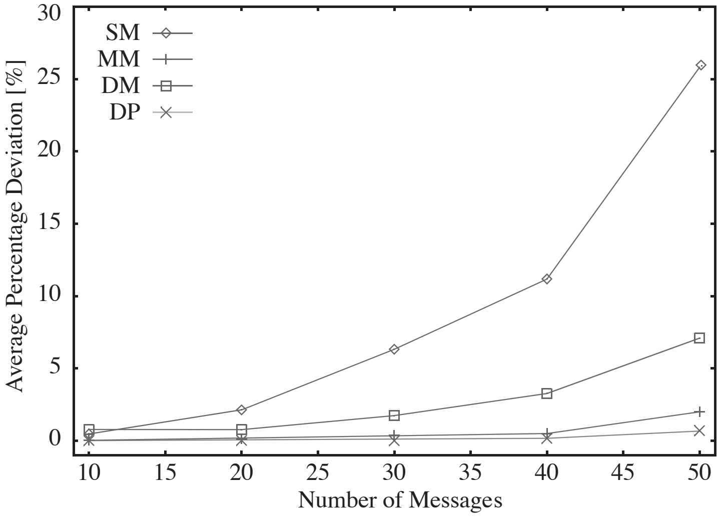


FIGURE 15.36
Four Approaches to Message Scheduling: The Influence of the Number of Messages

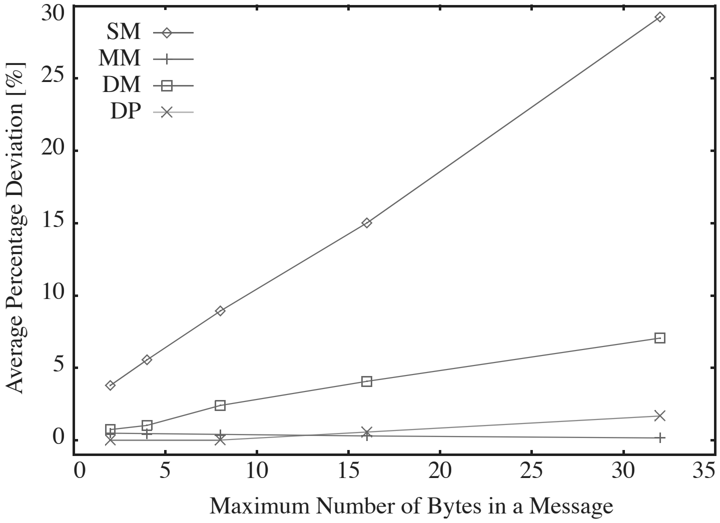


FIGURE 15.37
Four Approaches to Message Scheduling: The Influence of the Message Sizes

TABLE 15.2
Percentage deviations for the greedy heuristics compared to SA.

		80 tasks	160 tasks	240 tasks	320 tasks	400 tasks
SM	avg.	0.12%	0.19%	0.50%	1.06%	1.63%
	max.	0.81%	2.28%	8.31%	31.05%	18.00%
MM	avg.	0.05%	0.04%	0.08%	0.23%	0.36%
	max.	0.23%	0.55%	1.03%	8.15%	6.63%
DM	avg.	0.02%	0.03%	0.05%	0.06%	0.07%
	max.	0.05%	0.22%	0.81%	1.67%	1.01%
DP	avg.	0.01%	0.01%	0.05%	0.04%	0.03%
	max.	0.05%	0.13%	0.61%	1.42%	0.54%

creases, the difference between the approaches grows while the ranking among them remains the same. The same holds for the case when we increase the maximum allowed message size (Figure 15.37), with a notable exception: For large message sizes MM becomes better than DP, since DP suffers from the overhead due to its dynamic nature.

We were also interested in the quality of our greedy heuristics. Thus, we have run all the examples presented above using the greedy heuristics and compared the results with those produced by the SA based algorithm. Table 15.2 shows the average and maximum percentage deviations of the cost function values produced by the greedy heuristics from those generated with SA, for each of the graph dimensions. All four greedy heuristics perform very well, with less than 2% loss in quality compared to the results produced by the SA algorithms. The execution times for the greedy heuristics were more than two orders of magnitude smaller than those with SA. Although the greedy heuristics can potentially find solutions not found by SA, for our experiments, the extensive runs performed with SA have led to a design space exploration that has included all the solutions produced by the greedy heuristics.

The above comparison between the four message scheduling alternatives is mainly based on the issue of schedulability. However, when choosing among the different policies, several other parameters can be of importance. Thus, a static allocation of messages can be beneficial from the point of view of testing and debugging and has the advantage of simplicity. Similar considerations can lead to the decision not to split messages. In any case, however, optimization of the bus access scheme is highly desirable.

15.7 Configuration and Code Generation

Once the schedule has been created as described in Section 15.3, it is necessary to transform this schedule information into a device-specific configuration, so that the dedicated communication controller of the device knows what to do when. In TTP, this configuration is called Message Descriptor List (MEDL); different terms are used in other protocols. For brevity, we call it *communication configuration* throughout this section. The creation of such a communication configuration is described below in Section 15.7.1.

While the communication configuration is the most obvious configuration item, other parts of the system also need to be configured to be able to *process* it:

1. Middleware
    - COM layer
    - Potentially other layers, in case of a multilayer system (e.g., the AUTOSAR Basic Software Stack [19])
2. Application
3. Operating system (OS), if applicable

The creation of middleware configurations is described in Section 15.7.2. In addition, it is also possible (and often advantageous) to even generate the complete code of the middleware itself. This approach is discussed in Section 15.7.3. The application also needs some knowledge of the transmitted data, its structures and timing, and therefore requires a dedicated configuration for this specific purpose. If an operating system (OS) exists, it is also involved in the communication, and consequently also needs a configuration for its specific tasks. For brevity, all configurations needed in addition to the aforementioned communication and middleware configuration are called *third-party configurations* throughout this section. The creation of third-party configurations is described in Section 15.7.4.

15.7.1 Communication Configuration

The specific format and content of a communication configuration is hardware dependent. Each communication controller provides some specific features, and these features need to be configured correctly in order to bring the communication controller to work and interact with the other communication controllers on the network.

But not only differences in the hardware — or, more precisely, in the communication controllers — make it necessary to adapt a communication configuration on a per-node basis. Often, hardware buffers in the communication controller are very limited, but user requirements exist to provide the received frame at least for a certain amount of time (validity time span). One solution might be to copy all received frames from the hardware buffer to another location (e.g., an external RAM). But this solution is inefficient regarding execution time and resource usage. A better way is to only put those frames into buffers that are really needed by the specific host.

15.7.1.1 TTP — Personalized MEDLs

The cluster design defines the layout of rounds and cluster cycles, cluster modes, and the parameters required for clock synchronization, i.e., who *sends* what at what time.

It does not contain node-local information about the application data storage in the CNI of individual nodes. Each communication controller must have a personalized MEDL, which is derived from the cluster design. It contains node-local information and may contain special setup data required for internal purposes of specific communication controllers [340].

To optimize the CNI layout, a tool that has the node-local information, in particular the information about which messages a node *receives*, can customize the "abstract" MEDL and thus save execution time and buffer space: Only those messages really needed by the node are processed, stored and provided to upper layers and the application. Personalized MEDLs not only imply less processing work for the CPU that accesses the communication controller, but they also allow for a less strict timing of the tasks on that CPU. In addition, personalized MEDLs are usually smaller than "abstract" ones.

15.7.1.2 Monitor MEDL for TTP

However, one special node-level MEDL is created whenever MEDLs are made by the cluster design tool $^{\text{TTP}}$Plan: The Monitor MEDL. This MEDL is generated automatically right after scheduling, and is loaded into the communication controller of the Monitoring Node used for monitoring a TTP network. The Monitor MEDL has a special CNI message area layout that is required by the host software operating within the Monitoring Node. The node-level information of the Monitor MEDL does not interfere with node-level designs of the cluster; however, changes to the cluster design render the Monitor MEDL invalid.

15.7.1.3 Buffer Configuration for FlexRay

FlexRay controllers have configurable hardware buffers where data is written to and read from. In the AUTOSAR stack, this concept is abstracted toward the upper layers of the system: The FlexRay driver translates the hardware-specific (i.e., controller-related) information into the more abstract data of the upper software layers. For example, the FlexRay driver maps the information "which frame shall be received" to the corresponding registers of the FlexRay controller. In contrast to the CNI, which is available in TTP controllers, this buffer interface requires that the communication configuration is personalized, i.e., optimized with node-level information.

One part of the driver is the buffer configuration, which places each frame into its hardware buffer. Configuring the FlexRay driver thus generates the meta-level specification of what happens in the cluster. This requires the introduction of logical buffers, which are also known as "L-PDUs" in AUTOSAR. Such a buffer contains one — but not necessarily always the same — frame at any point in time. In FlexRay, there can be several configurations for a buffer, and even reconfiguration during runtime is possible. AUTOSAR, however, supports only one configuration per buffer. Depending on the type of controller, one such buffer corresponds to one or more hardware buffers (mapping in the generated code).

In FlexRay and AUTOSAR, PDUs (*Protocol Data Units*) are the central elements of data transmission. A PDU is a payload of information to be exchanged between different software layers on the node. In AUTOSAR, signals are not placed directly in frames, but in PDUs, which are handled by the PDU Router, see below.

15.7.2 Middleware Configuration

Once the hardware is configured, it is also necessary to configure the "upper" layers of the communication stack. While there may be other parts of middleware software which do not belong to the communication stack, in most systems the communication stack forms the largest and also most complex part. For example, in AUTOSAR [20], the communication stack consists of at least four, but up to seven, layers for a communication based on FlexRay:

- FlexRay Driver
- FlexRay Interface (FrIf)
- PDU Router
- COM Layer
- FlexRay NM (Network Management)
- FlexRay Transport Layer
- RTE (Run-Time Environment)

While some layers do not have many configuration parameters and thus are rather straightforward to configure, other layers — like the FlexRay Interface (FrIf) layer — imply the scheduling of send and receive tasks with respect to the timing and the validity span of the messages sent and received. As a representative of a rather complex layer, the FrIf layer is described in more detail in Section 15.7.2.2 below.

Another example are the communication layers for TTP. They directly access the TTP controller and provide an interface to the application. Figure 15.38 shows their architectural differences. Table 15.3 lists the main similarities and differences between these communication layers.

In contrast to the other layers listed there, the *fault-tolerant COM layer* (FT-COM) is completely generated by the $^{\text{TTP}}$Build design tool in order to optimize execution time and resource consumption. It operates closely together with TTTech's operating system $^{\text{TTP}}$OS. It supports packing and unpacking, reintegration (history state handling), byte order (endianness) handling, message agreement functions and handling of replicated redundant message instances. The FT-COM layer is described in more detail in Section 15.7.3.

The *table-driven COM layer* (TD-COM), the *hardware COM layer* (HW-COM) and the *high-speed COM layer* (HS-COM) are reusable engines that execute configuration tables generated by the design tool. These configuration tables define the messages that are sent and received by a specific node, and how to process them.

Both the HW-COM and the HS-COM layer decouple the TTP communication from the application functions, also in the time domain. They provide convenient,

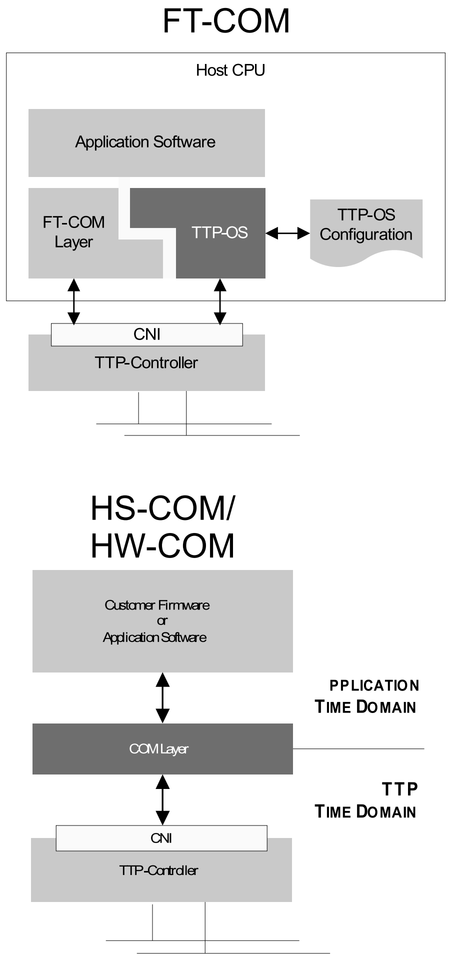


FIGURE 15.38
Examples of Different COM Layers

TABLE 15.3
COM layer properties compared.

Layer	FT-COM	TD-COM	HW-COM	HS-COM
Performance	++	+	+++	+++
Certification	none	DO-178B, level A certification for engine, verification tool for tables	DO-254 certification for IP model	DO-254 certification for IP model
Message sizes	1 to 32 bit, arrays, structured types	1 to 32 bit	32 bit only	32, 64, and 128 bit
Implementation	generated C code	C code, table-driven	VHDL code, table-driven	VHDL code, table-driven
Replication	yes	no	no	no
CPU Load	yes	yes	no	no
Asynchronous Access	no	yes	yes	yes

buffer-based interfaces to the application software. Their buffer interface allows for an easy mapping of ARINC 429 [10]. In addition, they are rather limited in their functionality as compared to the other layers presented. As a representative, the HS-COM layer is described in more detail in Section 15.7.2.3 below.

15.7.2.1 Configuration Format

Basically, there are two approaches to creating a middleware configuration:

- Source code, usually in C
- A binary block (memory area)

The C code actually comprises a big data structure, either a `struct` or simply an array, or any combination thereof. It might be generated just as a header file that is included in the main application code. In this case, it is automatically employed whenever the application is built. Otherwise it must be compiled and linked to the application in a separate step. As compilations are mostly done based on a Makefile, an additional file to be compiled is acceptable. The header file, which declares the data types used for the configuration structure in the C file, can be kept rather short.

An example is shown in Figure 15.41, representing a configuration for the HS-COM. Apart from the usual content of a C header file, it contains the declaration of the length of the configuration array and the array itself. The HS-COM configuration consists of 32-bit values only because they exactly match the size of an internal data access. This contributes to the high performance of the HS-COM. The comments in the table show the table index of the respective entry for easier navigation. More elaborate comments could be added if found beneficial, e.g., briefly describing each configuration parameter.

The advantages of the C code approach include the better readability and the fact that — due to prior compilation — only one file is present at runtime, which simplifies configuration management. If the configuration is not analyzed by a verification tool (see Section 15.8), good readability and means for easy navigation inside the (sometimes quite big) data structure can reduce certification efforts dramatically.

A binary block contains the configuration data in a structured form, so that the middleware directly and efficiently can access the individual parameters. It is interpreted by the middleware at runtime. Actually, the result of a compiled C code and a binary block may not differ at all for a certain configuration.

The advantages of a binary block include that it can be loaded separately from the application. If the development lifecycles of the application and the communication system are very different, or decoupling these two development tasks is advantageous for other reasons, the configuration can be generated and integrated into the system independently. A binary block needs to be loaded by the application and handed over to the middlware layer during the initialization phase.

15.7.2.2 FlexRay Interface Configuration

The FlexRay Interface (FrIf) layer is the part of the AUTOSAR communication stack that provides access to the FlexRay bus and its timing via the FlexRay Driver layer. Above the FrIf layer, there are the upper layers: PDU-Router (PduR) and FlexRay Transport Protocol (FrTp). The FrIf layer performs its actions according to the generated configuration. It is responsible for two basic tasks:

- It collects PDUs from the upper layers, packs the PDUs into frames and forwards the frames to the driver layer for sending on the FlexRay bus.
- It collects frames from the driver layer, unpacks the PDUs from the frames and forwards the PDUs to the corresponding upper layers (PduR or FrTp).

As can be seen from these characteristics, the FrIf appears PDU-based to the upper layers, but accesses the FlexRay bus in a frame-based fashion.

FrIf Actions

Receiving a frame starts when the FrIf receives the frame from the driver. The PDUs in the frame are unpacked, and the PDU data is passed to the corresponding upper layer (PduR or FrTp). This is done by calling the upper layer's respective API function, called *RxIndication* (receive indication). With this function, the PDU data is passed to the upper layer. After all PDUs have been processed, the frame reception is finished. Sending a frame starts with an upper layer (wanting to send a PDU) issuing a transmit request to the FrIf by calling the *FrIf_Transmit* API function. The FrIf stores every transmission request. It is important to note that a transmission request can occur at any point in the cluster cycle, unless the application is programmed to run synchronously with the FlexRay bus.

Later, when a frame is about to be transmitted, the FrIf checks each PDU in the frame, to see if its transmission has been requested. This point in time is determined during scheduling and can be influenced through the use of some of the advanced scheduling features described later in this chapter. For each PDU, the FrIf gets the PDU data that should be sent, packs the data into the frame and then sends the frame on to the FlexRay bus.

At some even later point in time, the FrIf confirms to the upper layer the transmission of each PDU by calling the *TxConfirmation* function. Again, this point in time is determined during scheduling. Through the use of this function, the upper layer can determine that a PDU has been sent.

For brevity, the receiving, sending and confirmation of a frame by the FrIf will in the following be referred to as *Actions*.

FrIf Job Handling

The sending and receiving of frames has to take place at predefined points in time as FlexRay is a time-triggered communication system. The timing is important for the following reasons:

- A received frame is only available for a limited time at the driver layer. If the FrIf misses the time window for getting the frame from the driver, the data of the frame might already have been overwritten and the frame data is lost. Note that the exact behavior in this situation is subject to the configuration, usage and number of the available buffers.

- If a frame is sent too late by the FrIf, the reserved bandwidth slot of the frame has already been transmitted by the driver, thus the current frame data cannot be sent. Depending on the setting of the corresponding parameter, the FlexRay controller sends either a Null frame or the current data from the frame buffer (which might be outdated).

The handling of actions at predefined points in time is implemented in the TTX-AUTOSAR FlexRay Stack by a hardware timer of the FlexRay module, which generates an interrupt each time a list of actions should be processed. A design tool with FrIf scheduling capability is responsible for calculating the timing of the actions. The output of the FrIf scheduler is called the FrIf schedule; it controls when an interrupt should occur, and which actions should be handled in a particular interrupt invocation. By accessing the compiled schedule, the FrIf layer coordinates its actions.

The main part of the schedule is the *JobList*, which is a collection of *Jobs*. There is only one JobList in the schedule. Each *Job* in turn is a collection of *Actions*; an action has an *action_type*, which can be either "*rx_frame*," "*tx_frame*" or "*tx_confirm*." The actions have already been described in the previous section.

A job stands for an invocation of the FlexRay interrupt on the target hardware. On the invocation of a particular interrupt, all the actions of the associated job are processed by the FrIf layer. The *job_activation_time* describes when the job's associated interrupt has to occur. The processing of jobs is done in the *FrIf_JobListExec*

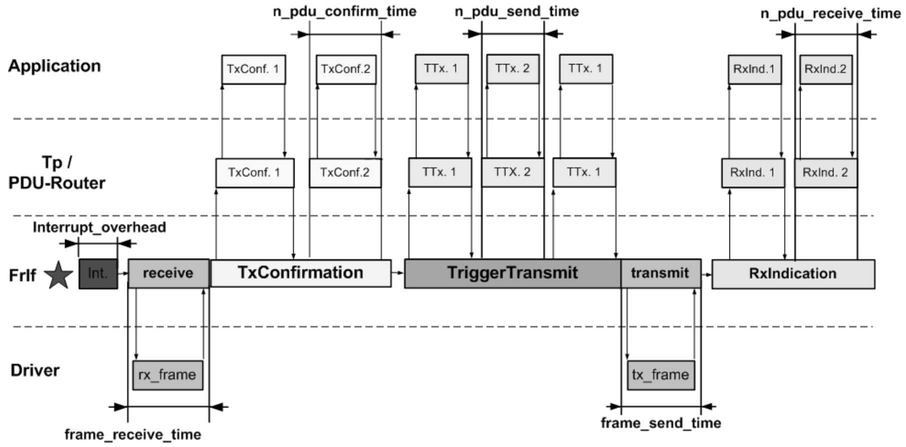


FIGURE 15.39
Sending and Receiving on FrIf Level

API function. This function has to be called in the interrupt service routine of the FlexRay interrupt. Figure 15.39 shows an example of a job and its actions.

Interrupt Overhead

The *activation_time* of the job is marked by a star in Figure 15.39. The delay between the *activation_time* and the actual processing of the first action (*rx_frame* in this case) is the *interrupt_overhead*.

The *interrupt_overhead* results from the fact that it takes some time for the CPU to get from the interrupt event into the *FrIf_JobListExec* function for the processing of the first action. Usually this time is very short. However, this is not always the case. Assume that an application needs to disable interrupts for a certain length of time, let's say $10\mu s$. If a FrIf interrupt occurs during this phase, the *FrIf_JobListExec* function is in the worst case processed after $10\mu s$ at the earliest, thus the *interrupt_overhead* needs to be configured accordingly.

Frame and Application Times

In order to put the FrIf actions into the FrIf jobs and to calculate the point in time for the interrupts, the time needed by each action must be known:

- The *frame_receive_time* is the time it takes the FrIf to receive a frame from the driver. It is defined as the time difference from the calling of the *frame_receive* function until this function returns.
- The *n_pdu_receive_time* is the time the FrIf needs to call the *RxIndication* function of the upper layer. The *RxIndication* function then passes the data to the upper layer.

- The *frame_send_time* is the time the FrIf needs to send one frame to the driver layer.
- The *n_pdu_send_time* is the time the FrIf needs to call the *TriggerTransmit* function of the upper layer. With the *TriggerTransmit* function, the upper layer passes the PDU data to be transmitted to the FrIf.
- The *n_pdu_confirm_time* is the time the FrIf needs to call the *TxConfirmation* function of the upper layer.
- The *frame_confirm_time* is not shown in the figure; it can be used to account for a constant overhead, which occurs during the processing of the *TxConfirmation* calls for all PDUs in the frame.

Using these definitions, a scheduler can calculate exactly how long the processing of a job will take (by summing up the *action times* for all *actions*). In the example from Figure 15.39, the execution time of the job can be computed with the following formula:

$$\begin{aligned} Duration &= interrupt\_overhead \\ &+ frame\_receive\_time \\ &+ (2 * n\_pdu\_receive\_time) \\ &+ frame\_send\_time \\ &+ (3 * n\_pdu\_send\_time) \\ &+ frame\_confirm\_time \\ &+ (2 * n\_pdu\_confirm\_time) \end{aligned} \tag{15.20}$$

Please note that this is the *worst-case execution time (WCET)* of the job. It may happen that the actual execution time for some invocations of this job on the hardware target is shorter; for example, if a received frame contains some PDUs which were not updated by the sending ECU. Then the FrIf does not need to call the *RxIndication* function for these PDUs, which results in a shorter runtime for this particular job invocation.

Figure 15.40 shows parts of a configuration for the FrIf layer. The major parts of the FrIf configuration are the definition of the PDUs as shown in the upper part of the figure, and the definition of the actions as shown in the lower part. The list of actions in this example contains 33 entries. Each entry specifies the type of the action, a reference to the frame, and a reference to the PDU. Further parts of the FrIf configuration (not shown) are the frame definitions, the definitions of the FrIf Jobs and JobLists, the action timing and the definitions of all used constants.

FrIf Schedulers

A FrIf scheduler may provide the user with advanced configuration options, such as “black-list” and “white-list” scheduling.

Black-list scheduling allows the user to specify *reserved_intervals* where no FrIf jobs may be scheduled. The intervals to be excluded from FrIf activity can be represented as a comma-separated list of ranges in microseconds. For example, setting

PDU Definitions

```
const ttx_frame_to_pdu_t _ttx_frame_to_frif_pdu_v_frame_0002_s [1] =
  { { PDU_ID_FRIF_fl_pdu_measure /* pdu_idx  */
    , 0  /* pdu_offset */
    , 8  /* pdu_len */
    , 1  /* use_update */
    , 17 /* updbit_bytepos */
    , 7  /* updbit_bitpos */
    , 0  /* is_tp_pdu */
    , PDU_ID_ROUTER_fl_pdu_measure /* destination_pdu_id */
    } /* [0] */
  };
```

FrIf Action Definitions

```
const ttx_frame_action_t _ttx_frame_action [33] =
  { { TTX_RX_AND_INDICATE /* action */
    , 1 /* frame_idx */
    , 18 /* mb_tutorial_web_018_a_r */ /* fr_pdu_id  */
    } /* [0] */
  , { TTX_TX_FRAME /* action */
    , 0 /* frame_idx */
    , 1 /* mb_tutorial_web_001_a_t */ /* fr_pdu_id */
    } /* [1] */
...
```

FIGURE 15.40

FrIf Configuration as C code — An Example the *reserved_intervals* to `0:100,5000:5100,10000:10100,15000:15100` means that FrIf jobs may not be scheduled during the first $100\mu s$ of the first four communication cycles, assuming a cluster cycle of $20ms$.

White-list scheduling provides the possibility to manually configure time intervals where actions for individual PDUs may be scheduled. The PDU-specific actions to be scheduled within a given time interval can be represented as a series of semicolon separated values according to the following format: “PDU/action/from:to.” For example, setting the *whitelist_spec* to `pdu_1/S/0:100;pdu_2/R/101:201` means that the send action for the PDU “pdu_1” can only be scheduled in the interval $0 - 100\mu s$ and the receive action of the PDU “pdu_2” can only be scheduled in the interval $101 - 201\mu s$.

Each interval of a white-list can be as large as the valid interval range or as small as the interval of the FrIf job in which the PDU action is to be scheduled, but not smaller. If the white-list spans more than one FrIf job, the user is in fact letting the scheduler choose which FrIf job to use for the processing of the action defined in the white-list. Furthermore, should the phase of the FrIf jobs vary between communication cycles, an analysis of this variation must be performed in order to ensure a large enough interval of the white-list to encompass suitable FrIf jobs in all communication cycles. More details on the configuration of the AUTOSAR communication stack for FlexRay, and especially of the FrIf, can be found in [341].

15.7.2.3 HS-COM Configuration

The HS-COM layer itself is a VHDL module that is part of an FPGA and provides the following features:

- Communication support for the AS8202NF TTP controller attached to an FPGA.
- Runtime and memory efficient packing and unpacking of messages to and from the TTP frames.
- Asynchronous access to the TTP data (buffering).
- Support for 128-bit event messages (i.e., queued best-effort transmission).

For optimization reasons, the HS-COM layer supports messages with a size of 32, 64 and 128 bits. It is further limited to the handling of message boxes whose size is an integer multiple of either 64 or 128 bits. A message box is a container that may hold one or several messages, but all messages in a message box must have the same size. The HS-COM can be executed in a highly efficient way as all these message types are aligned with the internal layout of the data registers.

Depending on the messages defined, the HS-COM acts differently:

- 64-byte messages will be assumed to be simple state messages, and the HS-COM will access the message box in 64-bit chunks.
- *Received* message boxes containing 32-bit messages will be accessed in 32-bit chunks and an additional 32-bit frame status (i.e., information whether the

Header file

```
#ifndef _HS_COM_h_
#define _HS_COM_h_ 1
#include "ptypes.h"
extern const ubyte4 hscom_config_len;
extern const ubyte4 hscom_config [];
```

C file containing the configuration

```
#include "ptypes.h"
const ubyte4 hscom_config_len = 32;
const ubyte4 hscom_config [32] =
  { 0x1         /*  [0] */
  , 0xc         /*  [1] */
  , 0x0         /*  [2] */
  , 0x0         /*  [3] */
...
  , 0x80000805L /* [29] */
  , 0x21        /* [30] */
  , 0x816       /* [31] */
  };
```

FIGURE 15.41
Communication Configuration as C code — An Example (HS-COM)

frame was received correctly) will be added to each message. With this feature, the content *and* the validity of a message can be retrieved in one action.

- Messages of type ‘128-bit’ are *always* treated as event messages and are accessed in 128-bit chunks. The queue depth for event messages is 32 FIFO entries each for sending and receiving, with 128 bits (i.e., one message) per entry.

The HS-COM layer performs a so-called *destructive read* when sending data on the bus, i.e., it sets the value of a read message to `0xFFFF...FFFF`. If the data in the send memory is not updated within a cluster cycle after reading, this value will be transmitted and tells the receiver that something went wrong, either with the transmission or with the send memory’s update. This mechanism prevents “old” data from being transmitted and mistaken for new.

Figure 15.41 shows an example configuration for the HS-COM layer. The HS-COM configuration consists of 32-bit values only for performance reasons. It comprises entries for the Register Area, the Pointer Area and the Command Area. In the shown example, the Register Area indicates the “SyncMode” to be `0x1`, and 12 lines to be used for the Pointer Area. The “host activity timeout” is set to 0. The last three shown lines represent commands (from the Command Area). Each command contains a parity bit in bit-position 31. Therefore, the first command starts with `0x8000`, and the two others start with 0. Bits 0 to 3 of each command specify the type of the command. For example, `0x80000805L` means to read one 128-bit event message starting from index 0. More details on the configuration of the HS-COM layer can be found in [342].

15.7.3 Code Generation

Middleware could be written by hand, and configured as discussed in Section 15.7.2. However, it is also possible to create the entire middleware layer with a design tool. Such automated creation can be exactly tailored to the communication needs of the schedule and the application, resulting in a highly optimized code. In the following, the *fault-tolerant communication (FT-COM) layer* for TTP is described in more detail as an example.

The FT-COM layer constitutes an interface between the communication services of the hardware, the operating system and the application software. According to Time-Triggered Architecture (TTA), each node executes an appropriate part of the distributed application, handling not only the data communication, but also the fault tolerance mechanisms designed for the system. As the FT-COM layer can be generated automatically by a design tool, the application code gets decoupled from the specific communication layer and fault tolerance mechanisms. This fact allows the application programmer to write source code that is highly reusable, easy to maintain, and transparent to many changes in the communication and fault tolerance design of the system. The FT-COM layer is generated as C source code for the node CPU, compiled and linked with the application code and executed on the same hardware as the application itself.

15.7.3.1 Feature Configuration

The FT-COM layer has several features that need configuration. A selection of these features is presented here, and relevant aspects regarding configuration and automatic code generation are discussed.

Subsystem Replication

A subsystem can be regarded as a set of tasks that take some input and produce some output. Each task is part of exactly one subsystem, but each subsystem may contain as many tasks as necessary. Several subsystems may be executed — independently of each other — on one host. A subsystem may also be executed simultaneously on more than one host (*replicated subsystem*). The first step toward fault tolerance can thus be achieved by replicating functionality, i.e., by replicating a subsystem.

The FT-COM needs to know how often a subsystem is replicated, and on which hosts these replicated subsystems run. It is expected that the FT-COM layer delivers a consistent view of the entire cluster regarding the value of a message, and provides diagnostic data to assess the “quality” of the provided data.

The Replica-Deterministic Agreement (RDA) Function

The receiver of a message m that is sent by a subsystem F, which is replicated with a replication degree of n, will in fact receive several message instances or *raw values* m_i of that message — one from each F_i that is active. But what is really wanted is the "correct" or "agreed" value. Therefore, the receiver needs to take the incoming instances m_i, run a function on them, and generate a single value m that will then be used for the application:

$$m = rda(m_1, m_2, \dots m_r) \tag{15.21}$$

The upper limit for r is the replication degree n, which applies when all replicas of F are active, the lower limit is zero. rda, the *agreement function*, must therefore be able to consistently handle an input vector of any length from zero to n. It must also be deterministic [258]. Several RDA functions exist and are selectable for the FT-COM, depending on the type of the subsystem (fail-safe or fail-consistent) [343]. Instead of encoding an algorithm that works for any n, it might yield a better performance to insert different implementations of the same algorithm into the FT-COM code, depending on n.

Application code that accesses the message m should never need to access the individual instances m_i, and can therefore be "ignorant" of the replication degree of the sender of m. A change of this replication degree only requires an update of the FT-COM layer, but not of the application itself.

Reintegration with H-State

Each (application) task generally takes some input, performs some function on it and produces a result as output; both input and output are messages. Furthermore, the task can contain static internal data that influences the computation and hence the output. The set of this internal data is called *h-state*.

For fast reintegration and enhanced robustness of the whole system, it might be necessary for a replicated instance of a subsystem to know this h-state of its partner instances. The network designer has to define a global message ("h-state message") that contains this information. Now the output can be considered solely a function of the input, no "hidden" data is involved anymore. For performance reasons, these h-state messages should only be received and processed when *no* valid h-state is currently present. The generated FT-COM layer needs to monitor the h-state, and to provide it when necessary.

Receiver Status

From the RDA mechanism, the *number of correctly received message copies* can immediately be derived by setting up a counter that is initialized with zero at the beginning of the message transmission interval, and increased by one for each message copy that is received correctly, finally giving r. This counter is called the *receiver status* of a message m. The receiver status is useful for several RDA functions. For example, the application software can use the receiver status to derive confidence information on how "good" m is. Another example is averaging: All valid m_i are summed up, and the result is divided by the receiver status. It would be incorrect to divide the sum by n, because in case of a failure of one or more replicas of F, the sum would contain less than n components.

In a programming language that treats the number zero as the Boolean equivalent of “false” and any number other (or at least greater) than zero as “true,” the receiver status can also be queried like a Boolean flag that yields “true” if the message is present, meaning that it was received correctly at least once and the RDA has yielded a result, and “false” if the message was not received correctly in this message transmission interval.

Sender Status

If an ECU hosts several subsystems, and one (fail-safe) subsystem fails, the others still should be able to send their data. Turning off the entire ECU is thus not an option. But as the communication controller works, it sends all messages, and potentially incorrect values for messages produced by the failed subsystem.

One classic strategy to handle this problem is to define an “invalid” value. This is unfavorable because it introduces a hidden information channel; if some application program fails to check for this special value in the right way, the system becomes inconsistent. Also, the “invalid” value might fall into the range of valid values after a software extension or upgrade. Any RDA function calculated in the FT-COM layer must take this into account.

The sender status of a message is part of the message itself, and therefore part of the input vector to the RDA function. The RDA will then consider a message that was correctly received, but has a sender status of “invalid,” to be non-present. Clearly, the FT-COM code performs better if the sender status is only considered for those messages actually having one, and no such code or if-statement exists for messages that have no sender status.

The receiver status of a message is generated at the receiver and is always available. Therefore, it can always be used for checking the availability of a message. But it does not carry the same amount of information that the sender status delivers: This information is generated by the sender, exists only if the system design requires it and allows the sender to explicitly *invalidate* the message contents while still sending the message; this can be necessary for a more complex node design where more than one subsystem is executed on the node.

The sender status implies additional effort for the sender, i.e., the FT-COM code generated for the sender. It must be updated, and additional bandwidth (even if only a single bit) is needed on the communication bus. Furthermore, the receiver must explicitly check this sender status, in addition to the receiver status that is always processed.

Message Timing

The message timing should not be done by the application software because, besides becoming unnecessarily complex, this could raise timing problems due to programming errors or faults during execution. Based on the separation of concerns, message timing should be handled by the FT-COM layer, which takes full responsibility and can be reused across different applications. The FT-COM code generator needs to respect all these timing constraints and "schedule" its tasks so that all messages are processed in time.

Message Buffer Handling

A replicated subsystem F that sends a message m may also want to receive this message. This sounds trivial, but requires some effort when replication is used, because in this case it is not correct to simply access the message in the local RAM.

Say N is a node where one of the replicas of F is executed, and assume that another subsystem G, which also runs on N, uses m as input. Receiving a message from a replicated subsystem requires an RDA (this is valid even for the subsystem that sends this message). Therefore, m exists twice on N: One instance is the value which is sent by F, to be entered in the RDA at all receivers (including N), the other one is the result of the RDA at N. Usually these will be equal, but if, for example, m is a sensor reading with an agreement function that computes the average, the local sensor may produce a slightly different result than the other redundant sensors in the system, and the value m that is actually used by the receivers (including G) is an average of all m_i that were transmitted in the previous round.

It follows that several message buffers can be required for a message, depending on whether the message is replicated or not:

- A *transmit buffer* for the message instance that is sent to all the receivers; this buffer is required for any message

- *Receive buffers* for each of the m_i

- A *result buffer* for the result of the RDA; this buffer is only required for replicated messages

Each of these buffers has the size (i.e., RAM requirements) of the message itself. A generated FT-COM may only provide all these buffers for messages where it is really needed, and save RAM if a message is not consumed by F or if the RDA function allows to directly use the sent value (e.g., "one-valid").

Packing of Bit Messages

Due to the CPU architecture of a node, the C variables containing the message values often use more memory than their data representation requires. The most common representatives of such a message type are Boolean messages, which have a data content of one bit, but are usually stored in a byte or even an `int`, depending on the CPU architecture and compiler.

However, since transmission bandwidth is rather expensive, the available net data rate should be optimally utilized. For this purpose, a Boolean message should be packed into a single bit, because it wastes a lot of space if it requires 16 or more bits for transmission. Similarly, a message which can take only one of 20 different values should not require 8 bits of transmission capacity, because the data content fits into

5 bits. Likewise, sensor data from an A/D unit that has a significant range of 10 bits does not need to be transmitted in a 16 bit word — but the packing algorithm needs to know which of the 16 bits are the 10 relevant ones.

At the receiver, the message needs to be expanded into a variable that is again easy to handle, like an `int`. The algorithm needs to be the exact inverse of the packing one, but must take into account several architectural properties that may differ between sender and receiver — the most prominent of all being the byte order.

On the other hand, the effort to efficiently (in terms of computation and code size) pack bit messages must be minimized, and there are much more efficient ways to achieve this than to simply consider the transmission buffer (frame) a long bit field and store all messages sequentially in this bit field. This is even true for standard messages of a size of 1, 2 or 4 bytes, and proper alignment can result in considerable performance gains.

However, manually programming such packing and unpacking routines for each bit message, and changing them consistently if something changes in the system specification (like a 10-bit A/D result being upgraded to 12 bits), is highly error-prone. Therefore, a layer that provides packing at the sender and unpacking at the receiver needs to be configured or created automatically, and must be supported by proper tools.

An automatically generated FT-COM layer may be optimized so that it only contains code that is really necessary for this particular platform, and that as many branches as possible are eliminated from the final code.

15.7.3.2 Implementation

The FT-COM layer must handle three major operations, specifically:

- Updating of the lifesign of the communication controller
- Packing of the messages into the proper frame buffers
- Unpacking and agreement calculation of all messages used by the application

All these operations are performed by special tasks (FT-tasks). One fundamental configuration option is the location of the frame buffers. If there exists a fast access to the CNI of the TTP controller, all packing and unpacking operations can be performed directly there. If not, it is more efficient to create local copies of the frames and to perform all operations locally. By setting this configuration option, the entire code can be created as it is best suited for the actual hardware.

Depending on the kind of the FT-task, it has to run either before or after an application task. The scheduler then has to ensure that the deadlines of the application tasks are met in any case. For all operations, the design tool needs to determine an interval within which the specific operation must be performed. To reduce task switching overhead, the design tool also should try to merge as many overlapping intervals as possible and to generate one FT-task for each of the resulting intervals; this leads to a minimal number of tasks.

The following sections describe how the schedule interval is determined for specific tasks.

Lifesign Update

The lifesign of the communication controller must be updated (by the host) at least once every round. In the pre-send-phase (the phase before the actual sending slot, d_{psp}), the controller checks if an update has been performed. Let $T_{s(n)}$ be the start of the controller's own sending slot of round n. The interval for the update of the lifesign in round n then is:

$$[T_{s(n-1)} \ldots T_{s(n)} - d_{psp}] \tag{15.22}$$

If the controller notices that the lifesign has not been updated, it goes into a passive state because there does not seem to be an application. Appropriate code for updating the lifesign has to be created and inserted into an FT-task that is scheduled for execution within this time interval.

Sender Tasks

The packing of messages into the proper frames in the CNI is done by sender tasks. The scheduling interval of these sender tasks must meet the following requirements:

- The latest possible *finish time* T_f for the packing of messages is the start of the pre-send-phase of the slot (i.e., start time of the slot minus the pre-send-phase).
- The earliest *task activation time* T_a is the time when the message is stable. This time is determined by the activation time of the application task plus its deadline. If at this point in time the message is not stable (i.e., the application task violates its deadline), the sender task must not start.

  If the task has a period that is different from the period of the message transmission on the network (defined in the cluster schedule), the activation instance leading to the shortest interval shall be considered, so that the latest value produced by the application is being sent over the network.

The interval $T_a \ldots T_f$ is computed for all messages sent by the application. All overlapping intervals should then be merged and a single FT-task should be generated, considering the runtime necessary for processing the messages and for updating the frames. To further reduce the number of required tasks, the sender tasks can also be merged with the lifesign tasks, if their intervals overlap and there is still enough runtime left for the lifesign updating.

Receiver Tasks

The receiver tasks must perform two operations; first the unpacking of the message instances (these instances will be used for the agreement), and then the computing of the specified agreement function. There are two different approaches to this:

- **Store and Process:** Unpack all message instances, store them in temporary buffers and perform the agreement function using the temporary buffers.

  The advantage of this approach is that it works with any kind of agreement (including majority voting) and also allows access to the individual raw values of the message.

  The disadvantage is increased memory demand: Every single message instance has to be stored.

- **Incremental:** Unpack only one message instance, perform the agreement on this instance, unpack the next message instance, . . . After all message instances have been agreed, the finalization of the agreement (e.g., divide the result by the number of values added to achieve the average) can be performed.

  The advantage is the lower memory consumption and often faster execution.

  The disadvantage is that it cannot be applied to all kinds of agreements, only to those which can be done sequentially. It must also be noted that in this case the raw values are not available to a diagnosis function at the receiver (usually not required).

The design tool can select the appropriate computation strategy for the selected agreement function, and then only insert this code into the receiver task. Dead or temporarily unused code can thus be avoided.

When it comes to optimization, it is not sufficient to just look at messages and message instances, but also their temporal distribution needs to be considered. Each time a periodically sent message is transmitted, this is called a message *generation*, not to be confused with a message *instance*. Consider a sender application that sends a specific message every $10ms$; further assume that this message is transmitted on two channels every $2ms$. This means that each message value generated by the sender is actually received 10 times at the receiver: Five different generations are received (one every $2ms$), and each generation contains two instances of the message.

In order to minimize the amount of global memory required by the FT-COM layer, it is necessary to perform the complete agreement for one message generation in a single FT-task. However, it is not always possible to pack all the steps of the complete agreement into a single task, since the individual (replicated) instances of a message generation may be spread throughout a whole round, and thus may have different and potentially non-overlapping validity spans. For the incremental approach, only some intermediate results need to be allocated globally if the agreement cannot be performed in a single task. Consequently, a good default is to use the incremental approach wherever possible.

For the receiver task generation, the validity interval of a message instance may be used as a possible scheduling interval. All overlapping intervals should then be merged and a single FT-task generated, considering the runtime necessary for the unpacking of the messages and for computing the agreement function.

To further optimize the memory footprint, the required RAM, and the execution time of the FT-COM layer, the design tool that creates the FT-COM code may filter out all message generations that are not used by application tasks. This can be done by comparing the activation times of the application tasks receiving the message with the validity intervals of the message generations. Only this reduced set of message generations will be retrieved from the network and provided to the application.

Code Generation

The TTP design tool $^{\mathrm{TTP}}$Build, which is available from TTTech, is able to automatically create FT-COM layer C code. $^{\mathrm{TTP}}$Build creates three files for each node (the names of these files are defaults and can be changed to any desired filename by the user):

- The message definition file `ttpc_msg.h` contains macro and variable declarations for the message buffers of incoming and outgoing messages on the specific node. This file, when included into application code, provides access to the message buffers, which are the only interface between the application program and the FT-COM layer. Function calls are not provided as they are not necessary for communication purposes.

  Some function-like C macros are offered to increase the readability of the generated code; for example,

  ```
  tt_Message_Status (temperature)
  ```

  is provided as a macro (looking like a function) to access the sender status of a message named `temperature`. In fact, the macro simply expands to the name of a variable, which is the message buffer containing the sender status of `temperature`, but the macro call improves the clarity of the statement. It will continue to work even if the implementation of the sender status should change in the future.

- The FT-COM layer C code is written to `ttpc_ftl.c` and comprises individual tasks called by the operating system (OS).

  The generated code is documented (the comments are also generated automatically, of course) to provide some insight into the workings of the FT-COM layer, but should never be changed manually. All changes will be lost when the code is generated again.

- `ttpos_conf.c` contains the configuration tables of the operating system, which tell the OS about the activation times and deadlines of all tasks on the node (application and FT-COM tasks alike). Although these tables are not part of the FT-COM layer, they are crucial for its proper operation, and are therefore also automatically generated by $^{\mathrm{TTP}}$Build.

  The contents of this file, although correct C code, are not intended to be human-readable, because they represent binary configuration data rather than program code (see option (a) in Section 15.7.2). As different operating systems require different formats, this file needs to be generated differently for each operating system that is supported by the design tool.

Additionally, a personalized MEDL can be generated to be loaded into the controllers of the host. This enables the definition of host-specific user interrupts and an optimized CNI layout.

15.7.4 Configuration of Third-Party Software

The operating system (OS), if one is present, and the application itself need to be configured, too. Design tools specifically designed to create communication configuration also need to interact with the development environment and configuration interfaces already available for the particular OS, the application or any other third-party software, e.g., a diagnostic module.

Typically, third-party software that interacts with the middleware, and in specific with the part that handles the communication, the communication stack, needs to know about a couple of things:

- **Layout and position of the messages:** The application must know by some means where the messages it reads and writes are located and how big they are. The most practical way is to have a memory-mapped interface. In this case, a C header file is required which contains `#define` statements. The application can refer to a certain message by name, and, based on the definitions in the header file, this name is mapped to a location in the memory. Of course, it is mandatory that the communication stack also has the same knowledge, but this is part of the communication configuration. Another possibility is to have a function call interface. Here, too, it is advantageous to have a mapping of message names (e.g., as defined in the design tool where all messages are specified) to certain IDs.

- **Interrupts:** The design of the communication stack may require the configuration of an interrupt for internal use in the communication stack. It might be helpful if any time a frame has been received by the hardware, a distinct interrupt is raised to indicate the arrival of the frame. Usually, this is a very high-priority interrupt. In the Interrupt Service Routine (ISR), the respective functions from the communication stack are called to handle this frame. These function calls must be registered beforehand, and the OS needs to know which interrupt to look at and to propagate. In addition, it may be possible to specify the interrupt priority level, the required stack size of the ISR and the vector to the service request register of the CPU.

- **Task properties and activation times:** If the communication stack is not interrupt-driven, it might need the activation of certain functions or tasks at certain times. Especially in a fully time-triggered environment, where the application and the OS are also synchronized to the communication network, this approach is favorable. As the OS dispatches tasks, it needs to know which communication task to start when, and with which assigned resources. In a real-time and time-triggered environment, the OS also needs to know the deadline of this communication task. Usually, one task is created for every message that has to be received or sent. For performance reasons, such tasks may be put together, forming task chains. For task chains, the OS needs to know similar properties as for tasks, in order to correctly interact with the communication stack.

- **Timer configuration:** If a timer is needed by the middleware, it has to be configured. All relevant details of this timer configuration also need to be part of the data that is shared between the middleware and the OS.

Development tools may generate parts of an OS configuration in the standardized *OSEK Implementation Language (OIL)* [242] format. As OIL comes in many vendor-specific flavors, it is very important to precisely determine the OIL version as well as the vendor-specific variant the generated file should have. The data can then be transferred to the OS by an OS configuration tool. In contrast to a complete OS configuration, a development tool for the communication stack may only provide the basic information necessary to run the various layers of the communication stack.

Development tools may also provide the relevant information as discussed above in other formats, e.g., in an XML-style fashion. Many operating systems come along with their own — and sometimes very specific — definition of the structure and possible content of OS configuration files. In such cases, either the development tool for the communication stack can be extended to write these files, or an additional conversion step needs to be introduced. A special-purpose tool or a self-written script may do the conversion job, too.

If the integration of the development tool and the OS is very good, the tool creates C files that fit the application. One C file may contain the message declaration and the type definition for every message that is sent or received by the application tasks. Another file may contain the configuration tables for the OS and comprise basic information on the respective node and task schedule. Ideally, the configuration tables for the OS are read, extended and then written back so that a configuration of a different origin is preserved. These configuration files are compiled and linked to the respective application to ensure the proper dependencies.

15.8 Verification

Society and law often request evidence that a particular system is fit for use and will not fail (or only in very rare cases), especially where the safety of humans is concerned. *Certification* by an accepted authority provides this kind of evidence; hence, most systems need to get certified for a particular use. For example, without permission granted by the Federal Aviation Administration (FAA), a commercial aircraft is not allowed to be operated in the US. Similar legal directives apply in other countries.

There exists a variety of certification standards, most prominently DO-178B [269] for software in aerospace, ISO 26262 [153] for automotive and

IEC 61508 [147] for industrial applications. For example, the FAA applies DO-178B for guidance to determine if the software will perform safely and reliably in an airborne environment [97].

Verification is one means listed in said standards to provide evidence for safe and reliable operation. To get a whole system certified, verification of certain artifacts that are part of the final system is hence necessary. As the schedule and the configuration items as described in Sections 15.3 and 15.7 are part of the final system, this need for verification applies to them. Details of the verification process and the area where verification is applicable are described in the respective standard.

To actually conduct verification, the use of tools is allowed and well established. Such tools are called *verification tools*. They need to be developed according to certain processes, also described in the standards mentioned above, and need to be *qualified* to be considered fit for their purpose. Tool qualification of verification tools is thus a crucial process on the way to getting a system certified.

In this section, we will discuss the impact of the different stages of verification on the software development process and the software itself, with the focus on the benefit of verification tools and their qualification.

The requirements for the verification of configuration items, imposed by certification standards, are discussed in Section 15.8.1. Various means to reduce cost during the verification process and related activities are presented in Section 15.8.2. A very prominent way is to use verification tools instead of manual verification performed in reviews. The verification of configuration items as well as the approach to use verification tools to assist the certification is presented in Section 15.8.3. Such verification tools must have a certain quality that can be reached by performing a tool qualification process. Details of this process and the implications posed on the development of the verification tools and the structure of the configuration items are also discussed there.

15.8.1 Process Requirements

In the aerospace industry, highly integrated safety-critical systems have been developed for decades. The FAA and other authorities have thus developed stringent certification requirements to meet the needs of the industry. Safety has always been the main focus of the system development. The regulations driving the safety of an aircraft are reflected in the Federal Aviation Regulations (FAR) 25 Paragraph 1309 (for the US) or — internationally spoken — in the Joint Aviation Regulations (JAR). For the methods of compliance with the FAR and JAR 25 requirements for a new system design, five methodologies are generally adopted, some of which are described in more detail in ARP 4754 [302] and ARP 4761 [303]:

1. Analysis including engineering analysis, stress analysis, system modeling and similarity modeling.

2. Failure analysis including FMEA (Failure Mode and Effects Analysis), FTA (Fault Tree Analysis) and safety analysis (including Functional Hazard Assess-

ment (FHA), (Preliminary) System Safety Assessment ((P)SSA) and Common Cause Analysis (CCA)).

3. Laboratory tests including component tests, qualification tests, system tests and tests on an integrated systems test rig.
4. Ground tests — On-aircraft ground tests.
5. Flight tests — On-aircraft flight tests.

Nowadays, the aerospace environment is strongly influenced by software certification authorities. The rapid increase in the use of software in airborne systems in the early 1980s resulted in a need for industry-accepted guidance for satisfying airworthiness requirements. DO-178, and subsequent revisions, have been written to satisfy this need and provide guidance for system software development. These certification requirements are illustrated with an overview of the DO-178B development process below.

The emergence of safety-critical x-by-wire systems in the automotive industry now leads to similar certification bodies and standards. Safety-related recommendations are already published, such as the MISRA guidelines [230] and recently the ISO 26262 standard. The latter has been derived from the Functional Safety standard IEC 61508 to better suit the needs in automotive electric and electronic systems. However, a mandatory certification authority for the hardware and software of automotive control units is not yet established. We believe that much benefit can be gained from the aerospace industry's certification experiences and recent activities to reduce certification costs of safety-critical systems [128].

Common to all safety standards is the ALARP principle [129]. ALARP stands for "as low as reasonably practicable" and means that the residual risk shall be as low as reasonably practicable. For a risk to be ALARP, it must be possible to demonstrate that the cost involved in reducing the risk further would be grossly disproportionate to the benefit gained. Adherence to state-of-the-art standards is widely accepted to be reasonably practicable.

15.8.1.1 DO-178B

DO-178 [269] was first published by the RTCA in 1980. It is intended to be used as a guideline for the software development and verification of airborne software systems. Since its first publication, the standard has been revised twice (DO-178A in 1985, DO-178B in 1992) and a third revision is ongoing (DO-178C).

DO-178B classifies software according to five assurance levels, rated by the criticality of the software functionality. Level A, the highest criticality level, is required for software whose anomalous behavior causes a catastrophic failure condition. Level E, the lowest level, is required for software whose anomalous behavior has no effect on the system's operational capacity. For each of the classification levels, DO-178B prescribes guidelines for the planning, development, verification, configuration management, software quality assurance, certification and maintenance of the system software.

DO-178B is a process oriented document; however, it does not prescribe the use of a particular lifecycle or structured methodology. This decision is left to the practitioner; however, the guidelines do require both the lifecycle model (with transition criteria) and the development methodology to be formally identified in the software plans and agreed with the certification authority, e.g., the FAA via a Designated Engineering Representative (DER).

DO-178B implies a requirements-driven development process. System requirements are decomposed into top level software requirements, which are in turn decomposed further into lower level requirements. This decomposition continues until module-level code can be directly implemented from the lowest level of requirements definition. In addition to design requirements driven by software requirements, derived design requirements are created to facilitate completeness of the software design. It follows that each element of the code base is traceable to a system-driven requirement or derived design requirement. Source code not directly traceable to requirements is strongly discouraged by the DO-178B guidelines. Such code is termed "dead code" and must be removed before certification. Deactivated code, that is, code utilized by the control unit but not exercised in application environment (e.g., manufacturing related code), is permitted, but only when the method of deactivation is proven and verified.

The verification activities recommended by DO-178B are also requirements-driven. The level of verification effort prescribed is once again proportional to the assigned software criticality level. Level A defines the most stringent verification process. Level E requires no verification of code or configuration items at all.

Level A development requires a full independent review of all of the verification artifacts, which consist of test cases and procedures. It also mandates that full structural coverage, including modified condition decision coverage (MC/DC), is achieved for all of the software. The generation of suitable test cases and expected results to yield such coverage drives much of the cost of level A development. Even outside the aerospace industry, testing and verification can account for as much as 40% to 70% of the total development effort [29, 111].

DO-178B also requires the adherence to strict configuration management practices. These practices require the practitioner to configure the entire software life cycle environment such that it can be reconstructed upon request. It also requires that software artifacts can be reproduced in their entirety from the configured data.

15.8.1.2 IEC 61508

IEC 61508 [147] is an international standard of rules applied in industry. It is titled "Functional safety of electrical/electronic/programmable electronic safety-related systems." The goal of functional safety is to use suitable methods to reduce the probability of dangerous errors to an acceptable level.

The safety categorization of a system is determined by the quantitatively defined probability of errors (see Figure 15.42). There, the categorization as seen by the other standards mentioned here is also shown, giving a comparison of the various levels.

The residual error rate of the data communication should not rise above the ac-

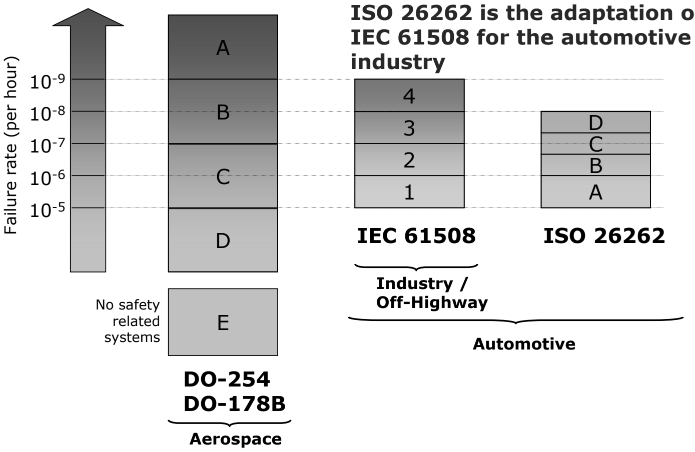


FIGURE 15.42
Comparison of Assurance Levels in Different Certification Standards

ceptable limit of 1% of the total errors of the system. For example, the following is valid for safety-relevant network signals in a SIL3 system according to IEC 61508: the probability of undetected corruption of such signals, which can lead to dangerous system errors, must be less than 10^{-9} per hour of operation (1% of the system error rate of $10^{-7}/h$). When a transmission error is detected, a corresponding system response must be triggered. In the case of a fail-safe system with only one communication bus, this means switching to a safe state; a fail-operational system must be able to transmit the data using an alternative transmission path.

IEC 61508 is less focused on requirements and, consequently, on verification of code and configuration items to satisfy these requirements. It is more concerned with functional safety: It is sufficient to show that the error rate is as low as requested, and that the system goes to a safe state in case of an error. Verification and thus the usage of verification tools is less commonly used, but may increase in the future.

15.8.1.3 ISO 26262

ISO 26262 [153] is an emerging norm for safety-relevant electrical and electronic systems in automobiles. It defines a process framework and process model together with required activities and work products, as well as applicable methods. The implementation of the norm is meant to guarantee the functional safety of an electrical/electronic system in a motor vehicle. The norm is derived from IEC 61508 specifically with regard to the domain of automobiles; compliance with this norm will tentatively start being mandatory in mid 2011 for all safety-relevant functions in motor vehicles.

In distributed control and regulating systems, data signals are transmitted over a network. The transmission route of such signals encompasses a sending device, one or more data buses (possibly including gateways), and one or more devices that receive and process the signals. Each of these components can, in case of errors, cause corruption, delay, loss, or repetition of the transmission or make it incorrect in some other way. To safeguard against errors in communication in vehicles, measures must be taken in order to detect errors during transmission. These errors must under no circumstances lead to a critical vehicle state.

ISO 26262 also contains lists of communication error classes that need to be dealt with, and measures that are known to be effective for recognizing these errors. In case a distributed system with safety requirements is being developed, in which safety-relevant data signals are transmitted over a network, it must be proven that all of these communication errors are detected reliably enough through effective mechanisms, so that the probability of an undetected communication error is below the required threshold. The calculation is done based on bit error rates of the network, the reliability of hardware (e.g., CRC units and RAM cells) and the applied methods (e.g., CRC polynomials and code word lengths).

15.8.2 Verification Best Practices

Verification is widely known as a time-consuming and costly activity. With the help of verification tools, costs can be reduced dramatically. But it is also necessary to obey a couple of best practices, so that the tools can be utilized best, and in as many steps as possible. In this section, we briefly present some best practices.

15.8.2.1 Reuse of Processes

Quality Assurance (QA) in aerospace is especially critical due to the relatively small production quantities and potentially large impact of failures on safety of operation. Accordingly, well-defined processes and many best-practice approaches exist. The development process for safety-relevant software development in the automotive sector can be derived from the time-tested development process for safety-relevant software development in aerospace. Since cost-effectiveness is a driving force behind innovation in the automobile industry, the efficient reuse of existing components is seen as one of the most effective factors in cost reduction. The savings of development and quality-assurance costs, as well as the robustness that results from time-tested and available components, contribute significantly to the realization of savings potential.

Due to the similar intentions of the above-mentioned standards, the development processes that are used for aerospace, automotive or industrial software development can be quite similar, too. It makes perfect sense to design a series of individual processes for all areas of business in an identical way.

- The processes for formal reviews and for change request management can be carried out with the help of tools that are uniform across the entire company in all areas of software development, and are carried out according to the same rules.

- The use of a common build framework for all lifecycle documents of software development makes it possible to simplify configuration management and to get an overview in the formal domain.

- A proven automatic certified test framework can be used to carry out several unit- and system-level tests in a particularly economical and exactly checkable manner.

15.8.2.2 Extending Checklists

QA stretches through the development, production, and operation and maintenance phases of an aircraft. During the development phase, a “verification and validation plan” has to be created in order to comply with any standard mentioned above. The plan contains checklists in addition to detailed descriptions of the checking procedures, test environments, test tools, documentation and result validation. These checklists are applied in the creation process of the corresponding lifecycle documents, as well as during their formal reviews.

The basis for project-specific checklists are the checklists from the standard software development process which usually exists in any company that develops safety-critical software. They are extended with the checklist points from the respective standard.

15.8.2.3 Use of COTS Products

When developing safety-related systems, testing indisputably causes the biggest overhead compared to development of conventional, non safety-related systems. However, the biggest savings potential also lies within this testing phase. A big proportion of testing time and cost can be saved when using off-the-shelf (COTS) components that are already safety-certified. Such components can either be complete units, or just sensors, or software modules like software drivers or protocol stacks. If these components or modules have already been tested by the supplier to the necessary degree required for the respective safety level, only the application layer and the interfaces to the COTS components need to be tested. The number of test cases for the application can be therefore significantly reduced.

Important prerequisites for the usage of COTS software modules are:

- A certificate indicating the safety integrity level of the component and the component failure rates that are needed for calculating the overall system failure rate.

- The availability of a safety manual that provides clear guidance on how to use the component in a safety-critical system.

The effort for the remaining required tests can be reduced by making use of appropriate tools for requirements management, configuration management, test execution, checking of coding standards, etc. However, all these tools also have to be qualified for use in safety-critical development.

15.8.2.4 Modular Certification

Modular certification according to DO-297 [270] is a rather new approach based on the need for certification of integrated modular avionics (IMA) and the corresponding system architectures [23]. The standard uses an architectural approach which enables the certification of small, reusable modules and applications. The needed functionality is established by connecting the single parts of the distributed application with a communication system. The standard breaks down the whole system into the following levels to map to the modular approach:

- **Module Acceptance:** A module is a component or a collection of components which may be software, hardware or a combination of both, which provides resources to the application and/or the system platform.
- **Application Acceptance:** An application is based on modules and performs a function.
- **System-level Acceptance:** The system level consists of one or several platforms which provide a computing environment, managing resources for at least one application. Furthermore, it establishes support services and platform-related capabilities like health monitoring and fault management.
- **Aircraft-level Acceptance:** The aircraft level considers the integration of the system into the aircraft and its systems.

Using such an architectural approach forces the reuse of legacy systems and provides the possibility of using modular platforms [175]. Therefore, the certification activities have to consider the certification of modules and especially their integration into the platform. An interesting approach considered for the future is to use formal methods to verify the integration.

The certification of single modules in this approach is fairly similar to the certification effort needed for reusable software components, i.e., for developing COTS products. Therefore, the reduction of certification effort applies here, too. In addition, the communication system which connects the modules needs to be fully approved.

15.8.2.5 Requirements Management

The requirements and design phases at the beginning are the most important parts of the software lifecycle process. The requirements define the expected output, and therefore need to be clear and easy to understand. The design is derived from the requirements and describes how they should be implemented. Every fault or obscurity in this phase has much impact later on. Requirements are the building blocks of the system. Therefore, the quality of the system depends on the quality of every single requirement.

Usually, the outcome of the design phase is stated in requirements, too. They are called low-level requirements, as opposed to high-level (software) requirements or system requirements, which are processed in the requirements phase. It is highly cost-efficient to apply the same processes for review and traceability checking on all requirements, rather than developing new processes for the design. Following this idea, the design thus consists of detailed low-level requirements, which mostly have to be traceable to the high-level requirements, and design components that describe complex algorithms and data structures to support the understanding.

Another major point is the traceability from the requirements to the design and further on to the test cases, down to the source code. This ensures that nothing is missing and everything has a reason for its existence. To ensure a constant quality level for the requirements and guarantee the traceability throughout the process, some basic points have to be recognized.

Tool Support

A database-centric requirements management tool provides a lot of advantages to the development and certification process. Firstly, several process steps are already included in the tool, hence the formal handling is simplified. Secondly, the waterfall-based top-down lifecycle process may be split up, which allows moving forward from requirements to implementation and verification without the need for consideration of other parts of the system. Furthermore, such a tool checks that all relevant traceability information is available. Additionally, some of these tools provide the possibility of creating evidence media which contain all necessary lifecycle and traceability information in an easy-to-review form. According to this efficient way to deal with the process, the effort for these steps may be reduced by about 20% with respect to the process necessary without tooling support.

Standardized Requirements Definitions

There should be standards for requirements definition, which provide guidelines for the development in order to facilitate their understanding. Furthermore, each requirement has to be self-contained because this supports the verification of each requirement.

Design Components

If a requirement describes a complex functionality, the developer should add definitions, figures and additional information which support the understanding. This encourages the demand for self-contained low-level requirements and helps to comprehend the whole system.

Testability

Each requirement has to be checked for testability. This has to be done by the requirements developer and especially by the reviewer. The easiest way to handle this is to write functional test cases in parallel to the requirements to find testability problems at an early stage of the requirements process. If this is not possible, the developer should at least give some advice or hints, regarding what to test, to the verification staff for efficient verification.

15.8.2.6 Test Vectors

In addition to the above, requirements-based test vectors (test cases and the input to automatic test procedures) can be automatically generated for each software product via a tool that is independent of the one used to generate the product code. These test vectors can cover nominal, MC/DC and robustness testing at the software module level. As with the code, these test vectors may have the requirements under test automatically inserted into them for better readability and traceability. All test vectors can then be parsed to create a complete test-vectors-to-requirements traceability matrix that is automatically inserted into the requirements management tool.

With different tools being used to generate test vectors and code, independence can be maintained, and therefore the test vector tool can be qualified as a verification tool as defined in DO-178B. With this qualification, peer reviews of the module (i.e., low-level diagram) tests are not required, resulting in a very large reduction of costs.

15.8.2.7 Test Suite

Another major concern regarding the verification process is the use of a test suite. The advantage of such a test suite is the possibility to automatically verify test cases and structural coverage. If the tool qualification package, which has to be provided to the authority, is already available for the chosen test suite tool, the verification effort may be optimized by about 10% compared to a process implementation without test suite tooling.

15.8.3 Verification Tooling Approach

The (automatic) generation of code and configuration items can be viewed as a step in the build process, similar to compilation. In a typical time-triggered communication system, these items can be grouped into three main blocks:

1. Communication configuration (i.e., MEDL) verification
2. Node-specific COM-layer verification
3. Application (control code) verification

This view eases the discussion about which processes shall be applied, and which measures and quality assurance metrics are applicable to source code generators and configuration generators. This view also implies, especially when applying DO-178B, that such generators are classified as development tools: It has to be shown that the output of said generators is correct with respect to the stated requirements, and that there is no code or configuration that is not covered by requirements. Tool verification is seen as less strict in the other mentioned standards; however, the considerations necessary for DO-178B form a valuable basis [63].

15.8.3.1 Output Correctness

To show evidence for output correctness, basically two different approaches are possible, and both are accepted and described in the standards. The first approach is to develop and test the generating tool in such a manner that for every possible input, the output is correct and adheres to the requirements stated in the input. Although such a development and certification of a code or configuration generator is costly, it removes the requirement to perform verification — often conducted by means of peer reviews — of the code or configuration itself. The tool is considered trustworthy. Thus, the one-time cost of certification is far less than the continual cost of performing verification of code and configuration.

The second approach is to develop the generating tool without respecting any processes. The tool might be non-deterministic, based on unreliable libraries or other components or even produce false output in some cases. It actually may "guess" the output. Obviously, the tool itself and thus its output cannot be considered trustworthy. But such freedom to choose any strategy to get to a possible solution allows for much more advanced algorithms and a higher chance to find a solution for a particular problem. In a subsequent, additional step — the verification — it has to be shown that for the *particular* given input, the output is correct with respect to the requirements stated in that input. It should be noted that if the output of the tool is verified, the tool can be used without qualification according to the standards. Such is the case for nearly all code generators and schedulers.

15.8.3.2 Manual vs. Automated Verification

Verification of the output can be done manually or automated. For manual verification, usually peer reviews are conducted, and checklists and a detailed process description for the reviewers exist. Manual verification can be cost-effective if done only once or only a few times. But the result of manual verification may depend on the assigned reviewers and their experience and expertise, and the result may not be exactly reproducible.

Automated verification pays off if the configuration data is expected to change several times during development. This is definitely the case when iterative development processes are used. It may also pay off if potential changes during the maintenance phase are considered, too. Verification can be done much faster if a verification tool exists. But also with other development processes, automated verification may be advantageous: The expertise of all involved persons gets cumulated in the verification tool, and is utilized in all subsequent versions of the tool. In addition, the result provided by the verification tool is exactly reproducible.

The largest portion of today's software costs is driven by the generation of the test cases and verification data. This is especially true for the development of verification tools. Verification data is required for each possible aspect of a configuration item. Verification data extend the test cases with input vectors and output vectors. The generation of verification data may also be automated, and the same requirements regarding tool qualification apply as for verification tools.

DO-178B classifies tools used during the development phase into two categories:

- **Development tools:** Tools whose output forms part of the airborne software and thus can introduce errors in the source code base (e.g., code generators).
- **Verification tools:** Tools that cannot introduce errors but may fail to detect

> them. For example, a static analyzer that automates a software verification process activity should be qualified if the function that it performs is not verified by another activity. Type checkers, analysis tools and test tools are other examples.

The use of verification tools is an interesting aspect of DO-178B. It provides the possibility of getting complex algorithms, like schedulers, easily certified. The verification tools have to verify the results of these algorithms, to prove their safe and deterministic behavior. Furthermore, a tool qualification package is needed for the verification tool, which provides confidence regarding the tool. The verification tool and its tool qualification package are mostly less expensive, if the verification for correctness has to be done several times, than to certify the development tool — containing the constructive algorithm — itself. Moreover, it is possible to hide intellectual property in the development tools, as their interior need not be assessed. Only the verification tool is assessed.

15.8.3.3 Qualification of Verification Tools

Tool qualification of verification tools is easier and thus more cost-effective than certification of development tools due to several aspects. Basically, it must only be shown that the tool does not accept any invalid, incomplete, incorrect or malicious code or configuration. However, the tool may (although not favorable) mark correct configurations as incorrect. In such a case, manual verification is necessary. Usually, such an incident results in an updated version of the verification tool, which is able to also handle this case correctly, as the intention of tool-based verification is to have no need for manual revision.

Any configuration that contains at least one element not having a matching requirement, or whose matching requirement implies another value, must be considered incorrect. Quite often, several requirements have an impact on the value of a certain output element. The verification tool does not need to tell the correct value of an output element — it is sufficient if it marks the element (or set of elements) as incorrect. The fact that the verification tool need not be constructive contributes to the cost-effectiveness of verification tools.

Another big advantage in the qualification of verification tools is the possibility to view the tool as black box. Internals need not be assessed. Consequently, there may be unused or even dead code inside the verification tool. It is not necessary to provide a detailed design and low-level requirements. Low-level test cases are not necessary, either. Only high-level requirements and the corresponding test cases are necessary. The total number of test cases and test vectors is thus significantly smaller than for the certification of a development tool. It is also possible to qualify a third-party tool, of which no internals are known. And it is further possible to qualify a tool just for a particular use case.

With the automation of requirements testing (i.e., the verification of the output generated by development tools, with respect to the requirements stated in the input to these tools), and MC/DC testing at the module level, the majority of (manual) testing emphasis can be directed at the system level, toward hardware-software integration and robustness testing. This results in a higher quality product, with reduced testing costs. At the system level there is limited automation because the testing requires system-level knowledge not captured in the software requirements. As such, these tests still need to be created and mostly also executed by hand. Consequently, an ideal process removes much of the manual work required to create safety-critical software, leaving the system and software design engineers to work at the system integration and test level, resulting in an overall product quality improvement.

The verification of MEDLs using $^{\text{TTP}}$Verify will be discussed next, followed by a discussion of the verification of the configuration of a certain COM layer, the $^{\text{TTP}}$TD-COM Layer.

15.8.3.4 $^{\text{TTP}}$Verify

$^{\text{TTP}}$Verify is a comprehensive tool for the verification of TTP cluster designs, based on MEDLs. A TTP cluster contains a number of hosts exchanging messages in a statically defined temporal pattern. Any TTP controller in the cluster has stored this temporal pattern in its MEDL. This MEDL defines the whole transmission behavior on the bus and the local CNI interface behavior to the host controller. $^{\text{TTP}}$Verify reads the MEDL files and verifies their integrity as well as their conformance to the TTP protocol. It is verified that the MEDLs belong to the same cluster and do not contradict each other. Some aspects of fault tolerance of the whole cluster are also checked.

The output of $^{\text{TTP}}$Verify is a file that is divided into chapters for better readability. To allow a condensed view of the verification results, the user can customize the report to his needs. But the user cannot influence the verification algorithms to avoid conditions where the tool may fail due to bad user input. The command file structure and the output file structure are especially designed to support automatization (e.g., for extracting specific data), since the purpose of $^{\text{TTP}}$Verify is to support and improve the verification process for TTP-based systems.

$^{\text{TTP}}$Verify automates the verification of the TTP schedule and the MEDLs where this schedule table is stored inside the TTP communication controllers. The correctness of this schedule is analyzed by $^{\text{TTP}}$Verify and the resulting report has to be checked by additional tools or manually. Therefore, it is necessary to allow for easy extraction of information by tools as well as to provide a human readable representation of this data. $^{\text{TTP}}$Verify is designed to specifically support safety-critical application software. Based on the Time-Triggered Architecture (TTA) and the TTP communication system, $^{\text{TTP}}$Verify supports distributed fault-tolerant hard real-time application software.

$^{\text{TTP}}$Verify not only verifies the correctness of MEDLs, it also provides information about a MEDL or the cluster schedule. $^{\text{TTP}}$Verify provides a summary for any verified controller, including scalar data (e.g., macrotick length, membership position) as well as different tables summarizing specific properties of a MEDL. This includes properties of all round-slots in any cluster mode which is provided for any controller type. Additional tables will be provided for specific controller-dependent properties. Different controller types will provide different types of properties that are reported. This controller data is not only informative for the user. It can also be used to manually verify issues that are beyond the scope of $^{\text{TTP}}$Verify (e.g., order of slots). Furthermore, if $^{\text{TTP}}$Verify detects a problem in a MEDL, the controller summaries may also be of help in finding the root cause behind the reported fault. These controller summaries are written in the report in the respective chapter of the MEDL.

$^{\text{TTP}}$Verify also provides a complete dump of the MEDL content in a human-readable form. This is necessary for verification activities that go beyond the scope of $^{\text{TTP}}$Verify, and allows a significant gain of productivity for these purposes.

15.8.3.5 $^{\text{TTP}}$TD-COM-Verify

The TTP Table-Driven Communication Layer ($^{\text{TTP}}$TD-COM Layer) is a static table-driven communication layer between the application and the TTP controller. It is designed for multiple TTP networks that are attached to one single CPU, and includes optimization for redundant messages. The $^{\text{TTP}}$TD-COM Layer is a static embedded library written in C, which is certified according to DO-178B.

As the name suggests, it is driven by configuration tables. These tables are usually generated by $^{\text{TTP}}$Build in C source code format, then compiled and linked into the embedded application, and will then reside in the ROM of the embedded target. Since this data influences the correct behavior of the embedded $^{\text{TTP}}$TD-COM code, the used configuration data needs to be verified. This is the main application of $^{\text{TTP}}$TD-COM-Verify.

What $^{\text{TTP}}$TD-COM-Verify is:

- A tool to verify the correctness of the provided configuration data, which is used by the embedded source code of the $^{\text{TTP}}$TD-COM Layer.
- A tool that checks the configuration data in binary form (as an S19 file, a Motorola-specific ASCII text encoding for binary data). This guarantees an end-to-end verification and no further need to verify a compilation or another transformation step.
- A tool that verifies the configuration data for integrity and consistency.
- A tool that verifies the configuration data for internal and global consistency against all participating hosts' configurations.
- A tool that verifies the correctness of scheduled user-interrupts.

What $^{\text{TTP}}$TD-COM-Verify is not:

- A verification tool which verifies the correctness of the embedded code.
- A verification tool which verifies the correctness of the code of the configuration generation tool.
- A WCET measurement tool for the given configuration data.
- A blackbox test of the embedded $^{\text{TTP}}$TD-COM code.

- A verification tool to check the C source code in any form (coding guidelines, correctness, etc.).

$^{\text{TTP}}$TD-COM-Verify reads a tool configuration file (in XML format), which on one hand contains switches for the tool behavior, and on the other hand the input file names of all other involved files. The latter include the requirement specification as an Interface Control Document (ICD), as well as the configuration tables and MEDLs to be verified. The configuration tables and MEDLs are read as S19 images together with unified map-files. In addition, $^{\text{TTP}}$TD-COM-Verify uses the MHL partition header files. Optionally, the worst-case execution times (WCETs) can be supplied to $^{\text{TTP}}$TD-COM-Verify to check the timing requirements.

Data Flow

Figure 15.43 shows the interaction between the development tools and the verification tools. On the left side, the standard TTP toolchain with $^{\text{TTP}}$Plan, $^{\text{TTP}}$Build and $^{\text{TTP}}$Load is shown, which finally results in several MEDLs and $^{\text{TTP}}$TD-COM Layer configurations, one for every host. These source files are compiled by a C-compiler chosen by the customer. They might be linked with the user application and the $^{\text{TTP}}$TD-COM embedded library. Finally, the linker has to provide an S19 file which serves as the verification input for $^{\text{TTP}}$TD-COM-Verify.

Additionally, the compiler provides a map-file mapping symbols to addresses inside the S19 image. Since every compiler has its own map-file format, $^{\text{TTP}}$TD-COM-Verify will only accept a unified XML-based map-file. In this map-file the MHB allocations — which are given in the `tt_tdc_application_data_mhb_alloc_*.h` files — are included as well. This map-file handling is shown in Figure 15.43 between the C-compiler and the `map.xml` file(s). It includes the process of converting a compiler-specific map-file and the MHB message allocations into a unified XML-format map-file `map.xml`. Several requirements ensure that this conversion is done correctly. For checking those requirements, an additional small verifying tool is provided.

The following arrows show activities which have to be done by the user:

- The arrows from the customer database requirement specification files (command file for $^{\text{TTP}}$Verify, tool configuration file for $^{\text{TTP}}$TD-COM-Verify and the ICD) show the responsibility of the user to define application requirements inside those files independently of input data to the tool chain.

- The dashed line between $^{\text{TTP}}$Verify and $^{\text{TTP}}$TD-COM-Verify illustrates the responsibility of the user to check if all MEDL requirements that are needed for $^{\text{TTP}}$TD-COM-Verify passed the tests correctly. Before $^{\text{TTP}}$TD-COM-Verify is allowed to be operated, the MEDLs need to be checked for internal and global consistency by $^{\text{TTP}}$Verify. To this end, $^{\text{TTP}}$Verify uses a special command file as input for a cross-check with application requirements. This command file is usually provided directly by the user.

Besides the MEDLs, $^{\text{TTP}}$TD-COM-Verify needs some further input. Similar to

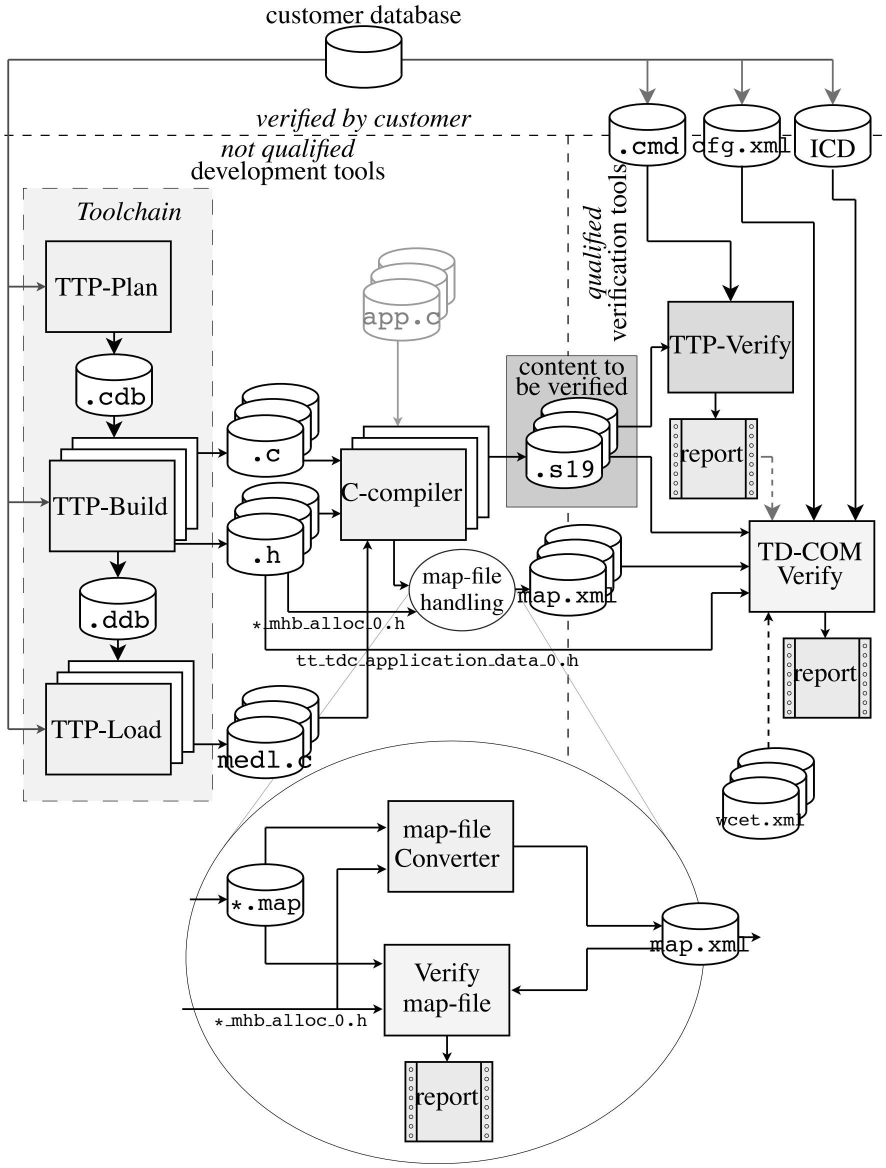


FIGURE 15.43
Interaction of the Development Tools with the Verification Tools

$^{\text{TTP}}$Verify's command file, the application requirements needed for $^{\text{TTP}}$TD-COM-Verify must be provided as ICD. While $^{\text{TTP}}$Verify supports verification for different cluster modes, the $^{\text{TTP}}$TD-COM Layer does not support cluster mode switches and has only one active cluster mode during the whole runtime. This active cluster mode needs to be provided to $^{\text{TTP}}$TD-COM-Verify through the ICD. The worst-case execution times (WCETs) of the frame copy tasks can be supplied to $^{\text{TTP}}$TD-COM-Verify in `wcet.xml` files. Every host needs a separate file. If these files are not present, $^{\text{TTP}}$TD-COM-Verify will just skip the timing requirements analysis.

Hence, a correct verification process would look like this:

1. Run $^{\text{TTP}}$TD-COM-Verify without WCET files to guarantee correctness of the binary table data.

2. If the tables are correct, use those tables to measure the WCETs of every frame copy task, and enter these times into the WCET files.

3. Rerun $^{\text{TTP}}$TD-COM-Verify with the newly created WCET files to check the scheduling timing requirements of the $^{\text{TTP}}$TD-COM Layer.

Certification Aspects

The host applications contain a number of high-level requirements for operation and interface to the $^{\text{TTP}}$TD-COM Layer. The $^{\text{TTP}}$TD-COM Layer has specific requirements for the proper delivery and retrieval of messages to/from the CNI. These requirements are composed of requirements derived from the $^{\text{TTP}}$TD-COM embedded code and the configuration tables. However, this procedure is very time consuming and expensive for large systems, and might slow down the development cycle dramatically. Therefore, a tool-based approach is considered. In such a tool-based approach, the interface requirements and the high-level requirements are provided as input to $^{\text{TTP}}$Build, which produces the code containing the C data structures used by the $^{\text{TTP}}$TD-COM Layer. $^{\text{TTP}}$Build and the C compiler suite are considered development tools according to the guidelines of DO-178B section 12.2, whereas $^{\text{TTP}}$TD-COM-Verify is considered a verification tool. $^{\text{TTP}}$TD-COM-Verify must examine the S19 images containing data from the configuration tables. Additionally, $^{\text{TTP}}$TD-COM-Verify has to validate them for correctness according to the application high-level requirements, the controller requirements and the $^{\text{TTP}}$TD-COM high-level and low-level requirements. By qualifying $^{\text{TTP}}$TD-COM-Verify in accordance with DO-178B, $^{\text{TTP}}$Build and the C compiler suite do not need to be qualified.

Bibliography

[1] A. Ademaj. Slightly-off-specification failures in the time-triggered architecture. In *Proc. of the 7th IEEE International High-Level Design Validation and Test Workshop*, page 7, Washington, DC, USA, IEEE Computer Society, 2002.

[2] A. Ademaj, H. Sivencrona, G. Bauer, and J. Torin. Evaluation of fault handling of the time-triggered architecture with bus and star topology. In *Proc. of the International Conference on Dependable Systems and Networks (DSN)*, pages 123–132, 22–25 2003.

[3] T. Amnell, G. Behrmann, J. Bengtsson, P.R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K.G. Larsen, M.O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL - Now, Next, and Future. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Modelling and Verification of Parallel Processes*, number 2067 in Lecture Notes in Computer Science Tutorial, pages 100–125. Springer–Verlag, 2001.

[4] E. Anceaume and I. Puaut. A taxonomy of clock synchronization algorithms. Research Report 1103, Institut National de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France, July 1997.

[5] E. Anceaume and I. Puaut. Performance evaluation of clock synchronization algorithms. Research Report 3526, Institut National de Recherche en Informatique et Systèmes Aléatoires (IRISA), Rennes, France, October 1998.

[6] C. Scheidler, P. Puschner, S. Boutin, E. Fuchs, G. Gruensteidl, and Y. Papadopoulos. Systems engineering of time-triggered architectures—the Setta Approach. In *Proceedings of the 16th IFAC Workshop on Distributed Computer Control Systems*, 2000.

[7] ARINC. *ARINC Specification 629: Multi-Transmitter Data Bus – Part 1: Technical Description*. Aeronautical Radio, Inc., Annapolis, MD, USA, November 1991.

[8] ARINC. *Backplane Data Bus. ARINC Specification 659.* Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, MD 21401, December 1993.

[9] ARINC. Multi-transmitter data bus: Part 1 technical description. arinc specification 629p1-5. Technical report, Aeronautical Radio Inc., Annapolis, MD, USA, March 31st 1999.

[10] ARINC. Arinc specification 429. digital information transfer system (DITS) parts 1,2,3. Standard ARINC 429, Aeronautical Radio Inc., 2001.

[11] ARTEMIS. The ARTEMIS strategic research agenda. `http://www.artemisia-association.org/sra`, 2006. [Online; accessed 25-August-2010].

[12] K. Arvind. Probabilistic clock synchronization in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):474–487, May 1994.

[13] Association for Standardisation of Automation and Measuring Systems (ASAM). *ASAM MCD-2 NET, Data Model for ECU Network Systems (Field Bus Data Exchange Format), Version 3.1.0*, 2009.

[14] Atmel Corporation. *AVR 308: Software LIN Slave*, May 2002. Application note available at http://www.atmel.com.

[15] Audi AG, BMW AG, DaimlerChrysler AG, Motorola Inc. Volcano Communication Technologies AB, Volkswagen AG, and Volvo Car Corporation. LIN specification and LIN press announcement. SAE World Congress Detroit, http://www.lin-subbus.org, 1999.

[16] Audi AG, BMW AG, DaimlerChrysler AG, Motorola Inc. Volcano Communication Technologies AB, Volkswagen AG, and Volvo Car Corporation. LIN specification v2.0, 2003.

[17] N.C. Audsley, I.J. Bate, and A. Grigg. The role of timing analysis in the certification of IMA systems. *IEEE Certification of Ground/Air Systems Seminar (Ref. No. 1998/255)*, Dept. of Comput. Sci., York Univ., London, UK, February 1998.

[18] Autosar. *AUTOSAR – Technical Overview V3.0*, 2006.

[19] Autosar. *General Requirements on Basic Software Modules, Release 3.1, Document Version 2.2.2*, 2008.

[20] Autosar. *List of Basic Software Modules, Release 3.1, Document Version 1.3.0*, 2009.

[21] A. Avizienis, J.C. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report 01-145, LAAS-CNRS, Toulouse, France, April 2001.

[22] O. Babaoglue and R. Drummond. (Almost) no cost clock synchronization. In *Proceedings of the* 7^{th} *International Symposium on Fault-Tolerant Computing*, pages 42–47, Pittsburgh, PA, USA, IEEE Computer Society Press, July 1987.

[23] A. Bahrami. Complex integrated avionic systems and system safety. In *Online Proceedings of the The Europe-US International Aviation Safety Conference*, 2005.

[24] M.B. Barron and W.F. Powers. The role of electronic controls for future automotive mechatronic systems. *IEEE/ASME Transactions on Mechatronics*, 1(1):80 –88, March 1996.

[25] G. Bauer and H. Kopetz. Transparent redundancy in the time-triggered architecture. In *Proc. of the Int. Conference on Dependable Systems and Networks (DSN 2000)*, New York, pages 5–13, June 2000.

[26] G. Bauer, H. Kopetz, and W. Steiner. The central guardian approach to enforce fault isolation in a time-triggered system. In *Proc. of the 6th Int. Symposium on Autonomous Decentralized Systems (ISADS 2003)*, pages 37–44, Pisa, Italy, April 2003.

[27] G. Bauer and M. Paulitsch. An investigation of membership and clique avoidance in TTP/C. In *Proc. of the 19th IEEE Symposium on Reliable Distributed Systems*, pages 118–124, 2000.

[28] G. Behrmann, A. David, K.G. Larsen, O. Müller, P. Pettersson, and W. Yi. UPPAAL - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[29] B. Beizer. *Software Testing Techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[30] R. Benesch. TCP für die Time-Triggered Architecture. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, June 2004. ARTEMIS

[31] C. Bergenhem and J. Karlsson. A process group membership service for active safety systems using tt/et communication scheduling. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 282 –289, December 2007.

[32] M. Bertoluzzo. Experimental activities on ttcan protocol. In *Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications, 2005. IDAACS 2005. IEEE*, pages 22 –27, 5-7 2005.

[33] P. Binns. A robust high-performance time partitioning algorithm. The Digital Engine Operating System Approach. In *Digital Avionics Systems Conference*. AIAA/IEEE, IEEE, 2001.

[34] P. Bishop. A methodology for safety case development. Technical report, Adelard, London, UK, 1998.

[35] P. Bjorn-Jorgensen and J. Madsen. Critical path driven cosynthesis for heterogeneous target architectures. In *Proceedings of the 5th International Workshop on Hardware/Software Co-Design*, pages 15–19. IEEE Computer Society, 1997.

[36] G. Bloor, G. Karsai, R. Reuter, S. Gulati, and S. Hutchings. The integration of anomaly, prognostics, and diagnostics reasoners to optimize overall vehicle health management goals. In *Proc. of the IEEE Aerospace Conference*, page 469, vol.2, 1999.

[37] B.W. Boehm, R. Madachy, and B. Steece. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2000.

[38] M. Borovicka. Design of a gateway for the interconnection of real-time communication hierarchies. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2003.

[39] BOSCH. CAN specification - version 2.0. available at http://www.bosch.de.

[40] J.D. Boskovic and R.K. Mehra. Multi-mode switching in flight control. In *Proc. of the 19th Digital Avionics Systems Conferences (DASC)*, pages 6F2/1 –6F2/8, vol.2, 2000.

[41] D. Bosnacki and D. Dams. Discrete-time promela and spin. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998.

[42] D. Bosnacki and D. Dams. Integrating real time into spin: A prototype implementation. In *FORTE XI / PSTV XVIII '98: Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*, pages 423–438, Deventer, The Netherlands, Kluwer, B.V., 1998.

[43] A. Bouajjani and A. Merceron. Parametric verification of a group membership algorithm. In *Proc. of the Symposium on Formal Techniques in Real-Time and Fault Tolerant System (FTRTFT), LNCS Vol. 2469*, pages pp. 83–105, Oldenburg, Germany, Springer-Verlag, September 2002.

[44] I. Broster, A. Burns, and G. Rodriguez-Navas. Comparing real-time communication under electromagnetic interference. 2004.

[45] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proc. of the 16th Annual International Conference on Automated Software Engineering (ASE 2001)*, pages 382 – 386, 26-29 2001.

[46] E.K. Burke and G. Kendall. *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Springer Verlag, 2005.

[47] D. Butler, T. Schmidt, and T. Waclawczyk. LIN protocol implementation using picmicro mcus. available at `www.microchip.com`, 2000. Microchip AN729.

[48] R.W. Butler, J.L. Caldwell, and B.L.Di Vito. Design strategy for a formally verified reliable computing platform. In *Proc. of the 6th Annual Conference on Computer Assurance (COMPASS) Systems*, pages 125–133, Gaithersburg, MD, USA, NASA Langley Res. Center, June 1991.

[49] G.C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer-Verlag New York Inc, 2005.

[50] I. Cardei, R. Jha, M. Cardei, and A. Pavan. Hierarchical architecture for real-time adaptive resource management. In *Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '00)*, pages 415–434, Secaucus, NJ, USA, Springer-Verlag New York, Inc., 2000.

[51] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. ARINC 659 scheduling: problem definition. *Real-Time Systems Symposium, 1994., Proceedings*, pages 165–169, December 1994.

[52] W.C. Carter. A time for reflection. In *Proc. of the 8th IEEE Int. Symposium on Fault Tolerant Computing (FTCS-8)*, page 41, Santa Monica, June 1982.

[53] CAST, Inc., IP Provider. LIN bus controller core, 2010. Available at `www.cast-inc.com/ip-cores/interfaces/lin/index.html`.

[54] CENELEC. *EUROPEAN STANDARD 50128: Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems*, March 2001.

[55] CENELEC. *EUROPEAN STANDARD 50159-1: Railway applications - Communication, signalling and processing systems; Part 1: Safety-related Communication in closed transmission systems*, March 2001.

[56] CENELEC. *EUROPEAN STANDARD 50159-2: Railway applications - Communication, signalling and processing systems; Part 2: Safety-related Communication in open transmission systems*, March 2001.

[57] CENELEC. *EUROPEAN STANDARD 50128: Railway applications - Communications, signalling and processing systems - Safety related electronic systems for signalling*, February 2003.

[58] P. Cholasta. LIN 2.0 mirror unit slave based on the MC68HC908EY16 MCU and the LIN 2.0 communication protocol. Application Note AN2885, Rev. 0, 11/2004, Freescale Semiconductor, 2004.

[59] G. Ciardo and C. Lindemann. Comments on ”analysis of self-stabilizing clock synchronization by means of stochastic Petri nets.” *IEEE Transactions on Computers*, 43(12):1453–1456, 1994.

[60] V. Claesson, H. Lönn, and N. Suri. An efficient TDMA start-up and restart synchronization approach for distributed embedded systems. *IEEE Transactions on Parallel and Distributed Systems*, 15(7), July 2004.

[61] D.D. Cofer and M. Rangarajan. Event-triggered environments for verification of real-time systems. In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, pages 915 – 922, vol.1, 7-10 2003.

[62] E.G. Coffman and R.L. Graham. Optimal scheduling for two-processor systems. *Acta Informatica*, 1(3):200–213, 1972.

[63] M. Conrad, P. Munier, and F. Rauch. Qualifying software tools according to ISO 26262. In *Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI*, 2010.

[64] FlexRay Consortium. FlexRay protocol specification ver. 2.1, 2005.

[65] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE micro*, 23(4):14–19, 2003.

[66] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Int. Computer Science Series, Addison-Wesley, second edition, 1994.

[67] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[68] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[69] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

[70] F. Cristian, H. Aghili, and R. Strong. Clock synchronization in the presence of omission and performance failures, and processor joins. *In Proc. of 16th Int. Symp. on Fault-Tolerant Computing Systems*, July 1996.

[71] F. Cristian and C. Fetzer. Fault-tolerant external clock synchronization. In *Proceedings of the* 15^{th} *International Conference on Distributed Computing Systems*, pages 70–77, Los Alamitos, CA, USA, IEEE, May 30–June 2 1995.

[72] P.H. Dana. Global Positioning System (GPS) time dissemination for real-time applications. *Real-Time Systems*, 12(1):9–40, January 1997.

[73] C.T. Davies. *Computing Systems Reliability*, Data Processing Integrity, pages 288–354. Cambridge University Press, 1979.

[74] L. de Moura, S. Owre, H. Rue, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In *Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, pages 251–254. Springer Berlin / Heidelberg, 2004.

[75] J.A. Debardelaben, V.K. Madisetti, and A.J. Gadient. Incorporating cost modeling in embedded-system design. *IEEE Design & Test of Computers*, 14(3):24–35, 1997.

[76] S. Dolev. Possible and impossible self-stabilizing digital clock synchronization in general graphs. *Real-Time Systems*, 12(1):95–107, January 1997.

[77] S. Dolev and J.L. Welch. Self-stabilizing clock synchronization with Byzantine faults. In *Proceedings of the* 14^{th} *ACM Symposium on Principles of Distributed Computing*, page 256. ACM Press, 1995.

[78] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona. The real Byzantine generals. In *Proc. 23rd Digital Avionics Systems Conf.*, volume 6.D.4, pages 61–11, October 2004.

[79] K. Driscoll and K. Hoyme. The airplane information management system: An integrated real-time flight-deck control system. *Real-Time Systems Symposium*, pages 267–270, December 1992.

[80] K. Driscoll, G.M. Papadoupoulos, S. Nelson, G.L. Hartmann, and G. Ramohalli. Multi-processor flight control system. Technical Report AFWAL-TR-84-3076, Honeywell Systems and Research Center, September 1984.

[81] K.R. Driscoll. Apparatus and method for fault detection on redundant signal lines via encryption. Patent U.S. 5307409, Honeywell, April 26th 1994.

[82] K.R. Driscoll. Apparatus and method for transmitting information between dual redundant components utilizing four signal paths. Patent U.S. 5386424, Honeywell, January 31st 1995.

[83] B. Dutertre and M. Sorea. Modeling and Verification of a Fault-Tolerant Real-time Startup Protocol using Calendar Automata. In *Proc. of the Joint Conference Formal Modelling and Analysis of Timed Systems (FORMATS), Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, Lecture Notes in Computer Science. Springer-Verlag, September 2004.

[84] C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.

[85] S.A. Edwards. *Languages for Digital Embedded Systems*. Springer Netherlands, 2000.

[86] P. Eles, A. Doboli, P. Pop, and Z. Peng. Scheduling with bus access optimization for distributed embedded systems. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems*, 8(5):472–491, 2000.

[87] W. Elmenreich. Time-triggered smart transducer networks. *IEEE Transactions on Industrial Informatics*, 2(3):192–199, 2006.

[88] W. Elmenreich and M. Delvai. Time-triggered communication with UARTs. In *Proceedings of the 4th IEEE International Workshop on Factory Communication Systems*, pages 97–104, 2002.

[89] W. Elmenreich, W. Haidinger, P. Peti, and L. Schneider. New node integration for master-slave fieldbus networks. In *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI 2002)*, pages 173–178, February 2002.

[90] W. Elmenreich and S. V. Krywult. A comparison of fieldbus protocols: LIN 1.3, LIN 2.0, and TTP/A. In *Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 747–753, 2005.

[91] C. Elmore. Electronic controls. *OEM Off-Highway*, November 2008.

[92] C. Engel, E. Jenn, P.H. Schmitt, R. Coutinho, and T. Schoofs. Enhanced dispatchability of aircrafts using multi-static configurations. In *Proc. of the Embedded Real Time Software and Systems*, Toulouse, France, May 2010.

[93] J. Erjavec and R. Scharff. *Automotive Technology: A Systems Approach*. Delmar Cengage Learning, 5th edition, 2009.

[94] J.A. Estefan. Survey of model-based systems engineering (MBSE) methodologies. *Incose MBSE Focus Group*, 25, 2007.

[95] FAA. Aviation databus assurance. Advisory Circular 20-156, Federal Aviation Administration, August 4th 2006.

[96] Federal Aviation Administration (FAA). Airworthiness directives; dassault model Falcon 2000ex and 900ex series airplanes. Airworthiness Directive Federal Register: (Volume 70, Number 39, Page 9853-9856), FAA, Docket No. FAA-2005-20425; Directorate Identifier 2005-NM-014-AD; Amendment 39-13987; AD 2005-04-15, March 1st 2005.

[97] Federal Aviation Administration (FAA). *Advisory Circular AC 20-115B*, 1993.

[98] C. Ferdinand and R. Heckmann. aiT: Worst-case execution time prediction by static program analysis. *Building the Information Society*, pages 377–383, 2004.

[99] M. Fernström and D. Ungerdahl. TTCAN Reference Application - An investigation on time-triggered network performance. Master's thesis, Chalmers University of Technology, 2006.

[100] C. Fetzer and F. Cristian. Lower bounds for function based clock synchronization. *In Proc. of 14th Int. Symp. on Principles of Distributed Computing*, August 1985.

[101] C. Fetzer and F. Cristian. An optimal internal clock synchronization algorithm. In *Proceedings of the* 10^{th} *Conference on Computer Assurance*, pages 187–196, Gaithersburg, MD, USA, IEEE, June 1995.

[102] C. Fetzer and F. Cristian. Integrating external and internal clock synchronization. *Real-Time Systems*, 12:123–171, March 1997.

[103] M. Fletcher. Progression of an open architecture: from Orion to Altair and ISS. Companion to report (Presentation) S65-5000-20-0, Honeywell, May 2009. FaultTolerant Spaceborne Computing Employing New Technologies 2009 Conference.

[104] FlexRay Consortium. FlexRay communications system – preliminary central bus guardian specification version 2.0.9. Technical report, BMW AG., DaimlerChrysler AG., Robert Bosch GmbH, and General Motors/Opel AG, 2002.

[105] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *FlexRay Communications System Protocol Specification Version 2.1*, May 2005.

[106] FlexRay Consortium. BMW AG, DaimlerChrysler AG, General Motors Corporation, Freescale GmbH, Philips GmbH, Robert Bosch GmbH, and Volkswagen AG. *Node-Local Bus Guardian Specification Version 2.0.9*, December 2005.

[107] A. Galleni and D. Powell. Consensus and membership in synchronous and asynchronous distributed systems. Technical report, 1995.

[108] GAMA. *ASCB: Avionics Standard Communications Bus Version C*. General Aviation Manufacturers Association (GAMA), Washington, DC, April 15 1996.

[109] M.-C. Gaudel, V. Issarny, C. Jones, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and F. Taiani. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, December 2002. Available as Research Report 54/2002 at http://www.vmars.tuwien.ac.at.

[110] M. Ghetie, H. Noura, and M. Saif. Fault diagnosis using balance equations methods and the algorithmic redundancy approach. In *Proc. of the 37th IEEE Conference on Decision and Control*, pages 586–591, vol.1, 1998.

[111] M. Ghiassi and K. I. S. Woldman. Dual programming approach to software testing. *Software Quality Journal*, 3(1):45–59, 1994.

[112] Robert Bosch GmbH. E-Ray FlexRay IP-module users manual revision 1.2.7, 2009.

[113] S. Godavarty, S. Broyles, and M. Parten. Interfacing to the on-board diagnostic system. In *Proc. of the 52nd IEEE Vehicular Technology Conference*, pages 2000 –2004, vol.4, 2000.

[114] S. Goldwasser, S. Micali, and R.L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, pages 281–308, April 1988.

[115] R. Gusella and S. Zatti. An election algorithm for a distributed clock synchronization program. In *Proc. of 6th Int. Conf. on Distributed Computing Systems*, pages 364–373, 1986.

[116] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by tempo in Berkeley UNIX 4.3BSD. *IEEE Trans. on Software Engineering*, 15(7):847–853, July 1989.

[117] J. C. Palencia Gutiérrez and M. González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real Time Systems Symposium*, pages 26–37, December 1998.

[118] J. C. Palencia Gutiérrez and M. González Harbour. Exploiting precedence relations in the schedulability analysis of distributed real-time systems. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 328. IEEE Computer Society, 1999.

[119] B. Heppner and H. Brauner. Assessment of whole vehicle behaviour by means of simulation. Technical report, Daimler AG, 2008.

[120] W. Haidinger and R. Huber. Generation and analysis of the codes for TTP/A fireworks bytes. Research Report 5/2000, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2000.

[121] B. Hall and K. Driscoll. A new aerospace network family. Presentation to INCOSE, Honeywell, October 2009.

[122] B. Hall, K.R. Driscoll, M. Paulitsch, and S. Dajani-Brown. Ringing out fault tolerance. A new ring network for superior low-cost dependability. *Dependable Systems and Networks, International Conference on*, 0:298–307, 2005.

[123] B. Hall, M. Paulitsch, and K.R. Driscoll. FlexRay BRAIN fusion: A FlexRay-based braided ring availability integrity network. *SAE World Congress, Paper No 2007-01-1492*, 2007.

[124] J.Y. Halpern, B. Simons, R. Strong, and D. Dolev. Fault-tolerant clock synchronization. In *Proceedings of the 3^{rd} ACM Symposium on Principles of Distributed Computing*, pages 89–102, 1984.

[125] F. Hartwich. *TTCAN IP Module - User's Manual*. Bosch, 1.6 edition, 11 2002.

[126] F. Hartwich, B. Müller, T. Führer, and R. Hugel. Timing in the TTCAN Network. Technical report, Robert Bosch GmbH, 2003.

[127] C. Haubelt, J. Teich, K. Richter, and R. Ernst. System design for flexibility. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 854–861. IEEE Computer Society, 2002.

[128] K. Hayhurst, C. Dorsey, J. Knight, N. Leveson, and G. McCormick. Streamlining software aspects of certification: Report on the SSAC survey. Technical report, NASA Technical Memorandum 1999-209519, August 1999.

[129] Health and Safety Executive (HSA). *Reducing Risks, Protecting People – HSEs Decision-Making Process*, 2001.

[130] M. Hecht, D. Tang, and H. Hecht. Quantitative reliability and availability assessment for critical systems including software. In *Proc. of the 12th Annual Conference on Computer Assurance*, Gaithersburg, MD, USA, June 1997.

[131] G. Heiner and T. Thurner. Time-triggered architecture for safety-related distributed real-time systems in transportation systems. In *Proc. of the Twenty-Eighth Annual Int. Symposium on Fault-Tolerant Computing*, pages 402–407, June 1998.

[132] R. Hexel. FITS: a fault injection architecture for time-triggered systems. In *Proc. of the 26th Australasian Computer Science Conference (ACSC '03)*, pages 333–338, Darlinghurst, Australia, Australian Computer Society, Inc., 2003.

[133] D. Höchtl and U. Schmid. Long-term evaluation of GPS timing receiver failures. In *Proceedings of the* 29^{th} *Precise Time and Time Interval Systems and Applications Meeting*, Long Beach, USA, December 1997.

[134] G.J. Holzmann. The model checker Spin. *Software Engineering, IEEE Transactions on*, 23(5):279 –295, May 1997.

[135] Honeywell. http://www.honeywell.com. accessed August 2010.

[136] K. Hoyme and K. Driscoll. SAFEbus. *IEEE Aerospace and Electronic Systems Magazine*, pages 34–39, March 1993.

[137] I. Hwang, S. Kim, Y. Kim, and C.E. Seah. A survey of fault detection, isolation, and reconfiguration methods. *IEEE Transactions on Control Systems Technology*, 18(3):636 –653, May 2010.

[138] IEC: Int. Electrotechnical Commission. *IEC 61508-7: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems – Part 7: Overview of Techniques and Measures*, 1999.

[139] IEEE. *Standard IEEE 802.4 – Information Processing Systems– Local Area Networks—Part 4: Token-Passing Bus Access Method and Physical Layer Specifications*, 1990.

[140] IEEE. IEEE standard 802.3 – carrier sense multiple access with collision detect (CSMA/CD) access method and physical layer. Technical report, IEEE, 2000.

[141] IEEE. *Draft Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems (V0.19.13)*. IEEE Press, New York, NY, USA, May 2002. IEEE Standard No. P1588; Product No. DS5905-TBR.

[142] IEEE. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. IEEE Press, New York, NY, USA, IEEE Standard No. 1588, March 2008.

[143] Aeronautical Radio Inc. *Avionics Application Software Standard Interface Part 1 – Required Services*, ARINC specification 653P1-2 edition, December 2005.

[144] Aeronautical Radio Inc. *Avionics Application Software Standard Interface Part 3 – Conformity Test Specification*, ARINC specification 653P-3 edition, October 2006.

[145] Aeronautical Radio Inc. *Avionics Application Software Standard Interface Part 2 – Extended Services*, ARINC specification 653P2-1 edition, 12 2009.

[146] National Instruments. FlexRay automotive communication bus overview. Technical report, August 2009.

[147] International Electrotechnical Commission (IEC). *IEC 61508: International Standard Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems*, 1998.

[148] International Standardization Organisation (ISO). *Road Vehicles – Controller Area Network (CAN) – Part 4: Time-Triggered Communication, ISO 11898-4*, 1993.

[149] International Standardization Organisation (ISO). *Road Vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication, ISO 11898*, 1993.

[150] International Standardization Organisation (ISO). *Road Vehicles - Controller Area Network (CAN) – Part 1: Data Link Layer and Physical Signalling, ISO 11898-1*, 1993.

[151] International Standardization Organisation (ISO). *Road Vehicles - Controller Area Network (CAN) – Part 2: High-Speed Medium Access Unit, ISO 11898-2*, 1993.

[152] International Standardization Organisation (ISO). *ISO/IEC 15765-3:2004 - Road Vehicles – Diagnostics on Controller Area Networks (CAN) – Part 3: Implementation of Unified Diagnostic Services (UDS on CAN)*, 2004.

[153] International Standardization Organisation (ISO). *ISO/DIS 26262: International Standard Road Vehicles – Functional Safety*, 2009.

[154] C. Jeffrey, N. Dumas, Z. Xu, F. Mailly, F. Azas, P. Nouet, R.J.T. Bunyan, D.O. King, H. Mathias, J.P. Gilles, and A.M.D. Richardson. Sensor testing through bias superposition. *Sensors and Actuators A: Physical*, 136(1):441–455, 2007. 25th Anniversary of *Sensors and Actuators A: Physical*.

[155] S.C. Johnson and R.W. Butler. Design for validation. *IEEE Aerospace and Electronic Systems Magazine*, 7(1):38–43, January 1992.

[156] H. Kantz and N. König. Tas control platform: A vital computer platform for railway applications. *Alcatel Telecommunications Review*, 2^{nd} Quarter 2004.

[157] H. Kantz and C. Koza. The ELEKTRA railway signalling system: Field experience with an actively replicated system with diversity. In *Proc. of the 25th International Symposium on Fault-Tolerant Computing (FTCS)*, pages 453 – 458, 27–30 1995.

[158] R. Kapeller. Design and implementation of a TTP/A master and gateway controller on a 32-bit microcontroller. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2001.

[159] J. Karlsson, J. Arlat, and G. Leber. Application of three physical fault injection techniques to the experimental assessment of the MARS architecture. In *Proc. of the 5th Annual IEEE International Working Conference on Dependable Computing for Critical Applications*, pages 150–161. IEEE Computer Society Press, 1995.

[160] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber. Integration and comparison of three physical fault injection techniques. In B. Randell, J. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 309–327. Springer Verlag, Heidelberg edition, 1995.

[161] S. Katz, P. Lincoln, and J. M. Rushby. Low-overhead time-triggered group membership. In *WDAG*, pages 155–169, 1997.

[162] B. Keinhuis, K. Vissers Deprettere, and P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *Proceedings of the 8th IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pages 338–350, 1997.

[163] K. Keutzer, S. Malik, A.R. Newton, J.M. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19(12):1523, 2000.

[164] M.S. Khan. Political and economic dimensions of Global Navigation Satellite System (GNSS). In *IEEE Proceedings of the Aerospace Conference*, volume 3, pages 3/1271 – 3/1276. IEEE, 2001.

[165] H. Kopetz. Event triggered versus time triggered. In *Proc. International Workshop on Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101. Springer Verlag, 1992.

[166] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. of 12th Int. Conference on Distributed Computing Systems*, Japan, June 1992.

[167] H. Kopetz. TTP/A – A time-triggered protocol for body electronics using standard uarts. In *International Congress and Exposition*, Detroit, MI, USA, The Engineering Society for Advancing Mobility Land Sea Air and Space, SAE International, February-March 1995.

[168] H. Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In *Proc. of the 5th IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*, Cheju Island, Korea, August 1995.

[169] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, 1997.

[170] H. Kopetz. Elementary versus composite interfaces in distributed real-time systems. In *Proc. of the Int. Symposium on Autonomous Decentralized Systems*, Tokyo, Japan, March 1999.

[171] H. Kopetz. *TTP/C Protocol – Version 1.0*. TTTech Computertechnik AG, Vienna, Austria, July 2002. Available at http://www.ttpforum.org.

[172] H. Kopetz. Fault containment and error detection in the time-triggered architecture. In *Proc. of the Sixth Int. Symposium on Autonomous Decentralized Systems*, April 2003.

[173] H. Kopetz. Time-triggered real-time computing. *Annual Reviews in Control*, 27(1):3–13, 2003.

[174] H. Kopetz. The fault-hypothesis of the time-triggered architecture. In *Proc. of the 18th Edition of the IFIP World Computer Congress*, August 2004.

[175] H. Kopetz. From a federated to an integrated architecture for dependable real-time embedded systems. In *Proceedings of the Eighth Annual High Performance Embedded Computing (HPEC) Workshop*, 2004.

[176] H. Kopetz. On the fault hypothesis for a safety-critical real-time system. In *Keynote Speech at the Automotive Software Workshop San Diego (ASWSD 2004)*, San Diego, CA, USA, January 10–12, 2004.

[177] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[178] H. Kopetz, G. Bauer, and S. Poledna. Tolerating arbitrary node failures in the time-triggered architecture. In *Proc. of the SAE 2001 World Congress*, Detroit, MI, USA, March 2001.

[179] H. Kopetz et al. The Time-Triggered Ethernet (TTE) design. In *Proc. of 8th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, May 2005.

[180] H. Kopetz and G. Grunsteidl. TTP-A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, 1994.

[181] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2):71–77, March 2001.

[182] H. Kopetz and R. Nossal. Temporal firewalls in large distributed realtime systems. In *Proc. of IEEE Workshop on Future Trends in Distributed Computing*, Tunis, Tunisia, IEEE Press, 1997.

[183] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 36(8):933–940, 1987.

[184] H. Kopetz and J. Reisinger. The non-blocking write protocol NBW: A solution to a real-time synchronisation problem. In *Proc. of the 14th Real-Time Systems Symposium*, 1993.

[185] J.M. Krause, M.J. Englehart, and D.A Shaner. Achievable performance of fault tolerant avionics clocks. In *AIAA Computing in Aerospace Conference, 8th*, Technical Papers. Vol. 2 (A92-17576 05-61), pages p. 608–622, Baltimore, MD, American Institute of Aeronautics and Astronautics, Oct. 21-24 1991.

[186] A. Krüger. *Interface Design for Time-Triggered Real-Time System Architectures*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 1997.

[187] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proc. of the IEEE*, 82:25–40, January 1994.

[188] L. Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[189] L. Lamport and P.M. Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the 3^{rd} ACM Symposium on Principles of Distributed Computing*, pages 68–74, 1984.

[190] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[191] L. Lavagno and C. Passerone. *Embedded Systems Handbook*, chapter 3, pages 3–1–3–22. CRC Press, 2006.

[192] M. Lebedev. GLONASS as instrument for precise UTC transfer. In *Proceedings of the* 12^{th} *European Frequency and Time Forum*, Warsaw, Poland, March 1998.

[193] E.A. Lee. Cyber physical systems: Design challenges. In *Proc. of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[194] P.A. Lee and T. Anderson. *Fault Tolerance Principles and Practice*, volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1990.

[195] G. Leen and D. Heffernan. Modeling and verification of a time-triggered networking protocol. In *Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies, 2006. ICN/ICONS/MCL 2006*, pages 178–178, 23-29 2006.

[196] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of 11th IEEE Real-Time Symposium*, pages 201–209, 1990.

[197] W. Lewandowski, J. Azoubib, and W.J. Klepczynski. GPS: Primary tool for time transfer. *Proceedings of the IEEE*, 87(1):163–172, January 1999.

[198] C. Li and S. Dey. Software-based self-testing methodology for processor cores. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(3):369 –380, March 2001.

[199] R. Lichtenecker. Terrestrial time signal dissemination. *Real-Time Systems*, 12(1):41–61, January 1997.

[200] LIN Consortium. LIN specification package revision 2.1, 2006.

[201] B. Liskov. Practical use of synchronized clocks in distributed systems. In *Proceedings of* 10^{th} *ACM Symposium on the Principles of Distributed Computing*, pages 1–9. ACM Press, 1991.

[202] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[203] C.D. Locke. Software architecture for hard real-time applications: cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[204] H. Lönn. Initial synchronization of TDMA communication in distributed real-time systems. In *19th IEEE Int. Conf. on Distributed Computing Systems*, pages 370–379, Gothenburg, Sweden, 1999.

[205] H. Lönn and J. Axelsson. A comparison of fixed-priority and static cyclic scheduling for distributed automotive control applications. In *Proceedings of the 11th Euromicro Conference on Real-time Systems*, pages 142–149. IEEE Computer Society Press, June 1999.

[206] H. Lönn and P. Pettersson. Formal verification of a TDMA protocol start-up mechanism. In *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*, pages 235–242, Taipei, Taiwan, IEEE, December 1997.

[207] T. Losert. *Extending CORBA for Hard Real-Time Systems*. PhD thesis, Vienna University of Technology, Institute of Computer Engineering, 2005.

[208] M. Lu, D. Zhang, and T. Murata. Analysis of self-stabilizing clock synchronization by means of stochastic Petri nets. *IEEE Transactions on Computers*, 39(5):597–604, 1990.

[209] J. Lundelius and N. Lynch. A new fault-tolerant algorithm for clock synchronization. In *ACM Symp. on Principles of Distributed Computing*, pages 75–88, 1984.

[210] J. Lundelius and N. Lynch. An upper and lower bound for clock synchronization. *Information and Control*, 62:190–204, 1984.

[211] J. Luo, K.R. Pattipati, L. Qiao, and S. Chigusa. Agent-based real-time fault diagnosis. In *Aerospace Conference, 2005 IEEE*, pages 3632–3640, 5-12 2005.

[212] S.R. Mahaney and F.B. Schneider. Inexact agreement: accuracy, precision, and graceful degradation. In *Proceedings of the* 4^{th} *ACM Symposium on Principles of Distributed Computing*, pages 237–249. ACM Press, 1985.

[213] S.M. Mahmud and A. Arora. Performance Analysis of Fault Tolerant TTCAN System. 2005.

[214] R. Maier, G. Bauer, G. Stoger, and S. Poledna. Time-triggered architecture: a consistent computing platform. *IEEE Micro*, 22(4):36–45, July/August 2002.

[215] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. Wiley, New York, 1990.

[216] G. Martin, F. Schirrmeister, and C.D.S. Inc. A design chain for embedded systems. *Computer*, 35(3):100–103, 2002.

[217] K. Marzullo and S. Owicki. Maintaining the time in a distributed system. In *Proceedings of the* 2^{nd} *ACM Symposium on Principles of Distributed Computing*, pages 295–305, 1983.

[218] K.A. Marzullo. *Maintaining the Time in a Distributed System: An Example of a Loosely Coupled Distributed Service*. PhD thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, USA, February 1984.

[219] M. McCabe, C. Baggerman, and D. Verma. Avionics architecture interface considerations between constellation vehicles. In *Proc. of the 28th Digital Avionics Systems Conference (DASC)*, pages 1.E.2–1 – 1.E.2–10. IEEE/AIAA, October 2009.

[220] M.D. Mesarovic and Y. Takahara. *Abstract Systems Theory*, chapter 3. Springer-Verlag, 1989.

[221] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[222] D. Michaud. *Maintenance Avionique - ATA 100 34 Test Automatique Bus Avionique Langage C*. Institut de Maintenance Aronautique, Universit Bordeaux I, 2006.

[223] V. Mikolasek, A. Ademaj, and S. Racek. Segmentation of Standard Ethernet Messages in the Time-Triggered Ethernet. Technical Report 22/2008, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2008.

[224] D.L. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

[225] P.S. Miner. Verification of fault-tolerant clock synchronization systems. Technical Report NASA Technical Paper 3349, NASA Langley Research Center, November 1993.

[226] R. Mores, G. Hay, R. Belschner, J. Berwanger, C. Ebner, S. Fluhrer, E. Fuchs, B. Hedenetz, W. Kuffner, A. Krüger, P. Lohrmann, D. Millinger, M. Peller, J. Ruh, A. Schedl, and M. Sprachmann. FlexRay – the communication system for advanced automotive control systems. In *Society of Automotive Engineers World Congress*, Detroit, MI, USA, SAE International. Document No 2001-01-0676, March 2001.

[227] M. Morgan. *The Avionics Handbook*, chapter Boeing B-777. CRC Press, Boca Raton, FL, USA, 2001.

[228] J. Morris, G. Lee, K. Parker, G.A. Bundell, and P.L. Chiou. Software component certification. *Computer*, 34(9):30–36, September 2001.

[229] MOST Cooperation, Karlsruhe, Germany. *MOST Specification Version 2.2*, November 2002.

[230] Motor Industry Software Reliability Research Association (MISRA). *Development Guidelines for Vehicle Based Software*, 1994.

[231] B. Müller, T. Führer, F. Hartwich, R. Hugel, and H. Weiler. Fault tolerant TTCAN networks. Technical report, Robert Bosch GmbH, 2002.

[232] C.J. Murray. Time-triggered protocol gains aerospace mileage. *EE Times*, September 2002.

[233] NXP Semiconductor. Fault-tolerant CAN/LIN fail-safe system basis chip. product data sheet, 2010. Available at `www.nxp.com/documents/data_sheet/UJA1061.pdf`.

[234] R. Obermaisser. CAN Emulation in a Time-Triggered Environment. In *Proc. of the 2002 IEEE Int. Symposium on Industrial Electronics (ISIE)*, volume 1, pages 270–275, 2002.

[235] R. Obermaisser. Message reordering for the reuse of CAN-based legacy applications in a time-triggered architecture. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 301–310, April 2006.

[236] R. Obermaisser and A. Kanitsar. Application of TTP/A for the Otto Bock Axon bus. Technical Report 27/2000, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, July 2000.

[237] R. Obermaisser and P. Peti. A fault hypothesis for integrated architectures. In *Proc. of the 4th Int. Workshop on Intelligent Solutions in Embedded Systems*, June 2006.

[238] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 2002.

[239] Object Management Group (OMG). *Smart Transducers Interface V1.0*, January 2003. Specification available at http://doc.omg.org/formal/2003-01-01 as document ptc/2002-10-02.

[240] A. Olson and K. Shin. Fault-tolerant clock synchronization in large multicomputer systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(9):912–923, 1994.

[241] OMG. Smart Transducers Interface V1.0. Available Specification document number formal/2003-01-01, Object Management Group, Needham, MA, U.S.A., January 2003. available at `http://doc.omg.org/formal/2003-01-01`.

[242] OSEK/VDX. *OIL: OSEK Implementation Language, Version 2.5*, 2004.

[243] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[244] J. C. Palencia, J. J. Gutiérrez Garcia, and M. González Harbour. On the schedulability analysis for distributed hard real-time systems. In *Proceedings of the Euromicro Conference on Real Time Systems*, pages 136–143, 1997.

[245] J.C. Palencia and M.G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26–37. IEEE Computer Society, 1998.

[246] M. Papatriantafilou and P. Tsigas. Self-stabilizing wait-free clock synchronization. In *Proceedings of the 4^{th} Scandinavian Workshop on Algorithm Theory*, volume 824 of *Lecture Notes in Computer Science*, pages 267–277. Springer-Verlag Berlin Heidelberg, Germany, July 1994.

[247] R.J. Patton. Fault detection and diagnosis in aerospace systems using analytical redundancy. In *IEEE Colloquium on Condition Monitoring and Fault Tolerance*, pages 1/1–120, 6 1990.

[248] M. Paulitsch and B. Hall. Insights into the sensitivity of the BRAIN (braided ring availability integrity network)–on platform robustness in extended operation. *Dependable Systems and Networks, International Conference on*, 0:154–163, 2007.

[249] M. Paulitsch and B. Hall. Starting and resolving a partitioned BRAIN. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:415–421, 2008.

[250] M. Paulitsch, J. Morris, B. Hall, K.R. Driscoll, E. Latronico, and P. Koopman. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. *Dependable Systems and Networks, International Conference on*, 0:346–355, 2005.

[251] P. Pedreiras and L. Almeida. Combining event-triggered and time-triggered traffic in FTT-CAN: Analysis of the asynchronous messaging system. In *Proc. of 3rd IEEE Int. Workshop on Factory Communication Systems*, September 2000.

[252] P. Peti, R. Obermaisser, and H. Kopetz. Out-of-norm assertions. In *Proc. of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 280–291, San Francisco, CA, USA, March 2005.

[253] P. Peti, R. Obermaisser, and H. Paulitsch. Investigating connector faults in the time-triggered architecture. In *Proc. of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA'06)*, pages 887 –896, 20-22 2006.

[254] P. Peti and L. Schneider. Implementation of the TTP/A slave protocol on the Atmel ATmega103 MCU. Technical Report 28/2000, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, August 2000.

[255] H. Pfeifer. Formal verification of the TTP group membership algorithm. In *Proc. of Formal Methods for Distributed System Development (FORTE XIII / PSTV XX 2000)*, pages 3–18. Kluwer Academic Publishers, 2000.

[256] H. Pfeifer, D. Schwier, and F.W. von Henke. Formal verification for time-triggered clock synchronization. In *Proc. of the 7th IFIP InternationalWorking Conference on Dependable Computing for Critical Applications (DCCA-7)*, pages 207–226, November 1999.

[257] M. Pfluegl and D. Blough. A new and improved algorithm for fault-tolerant clock synchronization. *Journal of Parallel and Distributed Computing*, 27:1–14, 1995.

[258] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6:289–316, 1994.

[259] S. Poledna. *Fault Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, Boston, 1996.

[260] P. Pop, P. Eles, and Z. Peng. Scheduling with optimized communication for time-triggered embedded systems. In *Proceedings of the Seventh International Workshop on Hardware/Software Codesign*, pages 178–182. ACM, 1999.

[261] P. Pop, P. Eles, and Z. Peng. *Analysis and Synthesis of Distributed Real-Time Embedded Systems*. Kluwer Academic Pub, 2004.

[262] P. Pop, P. Eles, and Z. Peng. Schedulability-driven communication synthesis for time triggered embedded systems. *Real-Time Systems*, 26(3):297–325, 2004.

[263] P. Pop, P. Eles, and Z. Peng. Schedulability-driven frame packing for multicluster distributed embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):140, 2005.

[264] P. Pop, V. Izosimov, P. Eles, and Z. Peng. Design optimization of time- and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems Volume*, 17(3):389–402, 2009.

[265] T. Pop, P. Eles, and Z. Peng. Schedulability analysis for distributed heterogeneous time/event triggered real-time systems. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings*, pages 257–266, 2003.

[266] T. Pop, P. Pop, P. Eles, and Z. Peng. Optimization of hierarchically scheduled heterogeneous embedded systems. In *Proceedings of 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 67–71, 2005.

[267] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1):205–235, 2008.

[268] D. Powell. Failure mode assumptions and assumption coverage. In *Proc. of the 22nd IEEE Annual Int. Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 386–395, Boston, USA, July 1992.

[269] Radio Technical Commission for Aeronautics, Inc. (RTCA). *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[270] Radio Technical Commission for Aeronautics, Inc. (RTCA). *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*, 2005.

[271] D. Ragan, P. Sandborn, and P. Stoaks. A detailed cost model for concurrent use with hardware/software co-design. In *Proceedings of the 39th annual Design Automation Conference*, pages 269–274. ACM, 2002.

[272] P. Ramanathan, K.G. Shin, and R.W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, October 1990.

[273] J.C. Ramirez and A.S. Piqueras. Learning Bayesian networks for systems diagnosis. In *Proc. of the Electronics, Robotics and Automotive Mechanics Conference*, volume 2, pages 125 –130, September 2006.

[274] Mathias Rausch. *FlexRay Grundlagen, Funktionsweise, Anwendung*. HANSER, 2008.

[275] C. R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems*. Blackwell Scientific Publications, 1993.

[276] FAST Report. Study of worldwide trends and r&d programmes in embedded systems. Technical report, 2005.

[277] RTCA. Software considerations in airborne systems and equipment certification. Standard DO-178B, RTCA, Inc., 1828 L Street, NW, Suite 805, Washington, DC 20036-5133, USA, December 1, 1992.

[278] RTCA. Design assurance guidance for airborne electronic hardware. Standard DO-254, RTCA, Inc., 1828 L Street, NW, Suite 805, Washington, DC 20036-5133, USA, April 19, 2004.

[279] RTCA. Environmental conditions and test procedures for airborne equipment. Standard DO-160E, RTCA, Inc., 1828 L Street, NW, Suite 805, Washington, DC 20036-5133, USA, December 9, 2004.

[280] B. Rumpler and W. Elmenreich. Considerations on the complexity of embedded real-time system design tasks. In *Proceedings of the IEEE International Conference on Computational Cybernetics 2006 (ICCC'06)*, pages 55–60, 2006.

[281] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999.

[282] J. Rushby. Systematic formal verification for fault-tolerant time-triggered algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, September 1999.

[283] J. Rushby. Formal verification of transmission window timing for the time-triggered architecture. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA 94025 USA, March 2001.

[284] J. Rushby. Modular certification. Technical report, Computer Science Laboratory SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, USA, September 2001.

[285] J. Rushby. An overview of formal verification for the time-triggered architecture. In *Proc. of the Symposium on Formal Techniques in Real-Time and Fault Tolerant System (FTRTFT), LNCS Vol. 2469*, pages 83–105, Springer-Verlag, Oldenburg, Germany, September 2002.

[286] J. Rushby and F. von Henke. Formal verification of the interactive convergence clock synchronization algorithm. Technical Report CSL-89-3R, Computer Science Laboratory, SRI International, CA, Menlo Park, USA, February 1989.

[287] SAE. ARP 5107 (aerospace recommended practice). guidelines for time-limited-dispatch analysis for electronic engine control systems. Technical Report Rev. B, Society of Automotive Engineers, November 2006.

[288] I. Saha and S. Roy. A finite state analysis of time-triggered CAN (ttcan) protocol using Spin. In *Computing: Theory and Applications, 2007. ICCTA '07. International Conference on*, pages 77 –81, 5-7 2007.

[289] I. Saha, S. Roy, and K. Chakraborty. Modeling and verification of TTCAN startup protocol using synchronous calendar. In *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, pages 69 –79, 10-14 2007.

[290] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2, 1984.

[291] A. Sangiovanni-Vincentelli. Electronic-system design in the automobile industry. *IEEE Micro*, 23(3):8–18, 2003.

[292] A. Schedl. *Design and Simulation of Clock Synchronization in Distributed Systems*. Doctoral thesis, Institut für Technische Informatik, Technische Universität Wien, Treitlstr. 1-3/3/182-1, Vienna, Austria, April 1996.

[293] F. Scheler and W. Schröder-Preikschat. Time-triggered vs. event-triggered: A matter of configuration? In *Proc. of the Workshop on Model-Based Testing*, Nürnberg, Germany, 2006.

[294] U. Schmid. Orthogonal accuracy clock synchronization. *Chicago Journal of Technical Computer Science*, 2000(3):3–77, August 2000.

[295] U. Schmid and K. Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12:173–228, March 1997.

[296] F.B. Schneider. A paradigm for reliable clock synchronization. Technical Report TR86-735, Computer Science Department, Cornell University, February 1986.

[297] F.B. Schneider. Understanding protocols for Byzantine clock synchronization. Research Report 87-859, Department of Computer Science, Cornell University, Ithaca, NY, USA, August 1987.

[298] W. Schwabl. *Der Einfluss zufälliger und systematischer Fehler auf die Uhrensynchronisation in verteilten Echtzeitsystemen*. Doctoral thesis, Institut für Technische Informatik, Technische Universität Wien, Treitlstr. 1-3/3/182-1, Vienna, Austria, October 1988.

[299] K.G. Shin and R. Ramanathan. Clock synchronization of large multiprocessor systems in the presence of malicious faults. *IEEE Transactions on Computers*, 36(1):2–12, 1987.

[300] O. Sinnen. *Task Scheduling for Parallel Systems*. Wiley-Blackwell, 2007.

[301] H. Sivencrona, P. Johannessen, M. Persson, and J. Torin. Heavy-ion fault injections in the time-triggered communication protocol. In *Dependable Computing, Lecture Notes in Computer Science*, volume 2847/2003, pages 69–80. Springer Berlin/Heidelberg, 2003.

[302] Society of Automotive Engineers (SAE). *ARP 4754: (Aerospace Recommended Practice) - Certification Considerations for Highly Integrated or Complex Aircraft Systems*, 1996.

[303] Society of Automotive Engineers (SAE). *ARP 4761: (Aerospace Recommended Practice) - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, 1996.

[304] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[305] W. Steiner. *Startup and Recovery of Fault-Tolerant Time-Triggered Communication*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2004.

[306] W. Steiner. TTEthernet Executable Formal Specification. Research report, 2009. Available at http://www.ttagroup.org/.

[307] W. Steiner. An Evaluation of SMT-based Schedule Synthesis For Time-Triggered Multi-Hop Networks. In *RTSS'10: Proceedings of the 31st IEEE Real-Time Systems Symposium*. IEEE, 2010.

[308] W. Steiner. Synthesis of Static Communication Schedules for Mixed-Criticality Systems. In *AMICS 2011: Proceedings of the 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*. IEEE, 2011.

[309] W. Steiner, G. Bauer, B. Hall, M. Paulitsch, and S. Varadarajan. TTEthernet dataflow concept. In *NCA*, pages 319–322, 2009.

[310] W. Steiner and B. Dutertre. SMT-Based formal verification of a TTEthernet synchronization function. In *FMICS*, pages 148–163, 2010.

[311] W. Steiner and W. Elmenreich. Automatic recovery of the TTP/A sensor/actuator network. In W. Elmenreich, editor, *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems*, pages 25–37, 2003.

[312] W. Steiner and H. Kopetz. The startup problem in fault-tolerant time-triggered communication. *International Conference on Dependable Systems and Networks (DSN 2006)*, June 2006.

[313] W. Steiner and M. Paulitsch. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. In *Proc. of the International Conference on Distributed Computing Systems*, pages 329–336, 2002.

[314] W. Steiner, M. Paulitsch, and H. Kopetz. The TTA's approach to resilience after transient upsets. *Real-Time Syst.*, 32(3):213–233, 2006.

[315] K. Steinhammer. *Design of an FPGA-Based Time-Triggered Ethernet System*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[316] K. Steinhammer, P. Grillinger, A. Ademaj, and H. Kopetz. A Time-Triggered Ethernet (TTE) switch. In *Proc. of Design, Automation and Test in Europe*, Munich. Germany, March 2006.

[317] J. Stelzer. LIN bus emerging standard for body control apps. *EE Times Asia*, September 2004.

[318] S. Subbiah and S. Nagaraj. Issues with object orientation in verifying safety-critical systems. In *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*, pages 99 – 104, 14-16 2003.

[319] Sunplus Technology Co., Ltd. LIN bus master note application using UART module. available at `mcu.sunplus.com`, 2006. V1.3.

[320] J. Swingler, J.W. McBride, and C. Maul. Degradation of road tested automotive connectors. *IEEE Transactions on Components and Packaging Technologies*, 23(1):157–164, March 2000.

[321] Systems Integration Requirements Task Group, Society of Automotive Engineers. *ARP 4754: Certification Considerations in for Highly-Integrated or Complex Aircraft Systems*, April 1996.

[322] Systems Integration Requirements Task Group, Society of Automotive Engineers. *ARP 4761 (Aerospace Recommended Practice) - Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment*, December 1996.

[323] B. Tabbara, A. Tabbara, and A. Sangiovanni-Vincentelli. *Function/Architecture Optimization and Co-Design of Embedded Systems*. Springer Netherlands, 2000.

[324] C. Tanzer. TTPos - the time-triggered and fault-tolerant RTOS. In *Real-Time Magazine* 99-4, 1999.

[325] Time-Triggered Technology TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP-Load: The Download Tool for the Time-Triggered Protocol – Version 6.1.6*, 2004.

[326] Time-Triggered Technology TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTP Bootloader: User Manual*, November 2005.

[327] K. Tindell and H. Hansson. Babbling idiots, the dual-priority protocol, and smart can controllers. In *Proceedings of the 1st Int. CAN Conference*, 1994.

[328] K. W. Tindell. Adding time-offsets to schedulability analysis. Technical Report YCS 221, Department of Computer Science, University of York, January 1994.

[329] K. W. Tindell, A. Burns, and A. J. Wellings. Allocating hard real-time tasks: an np-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[330] K. W. Tindell and J. Clark. Holistic schedulability analysis for distributed real-time systems. *Euromicro Journal on Microprocessing and Microprogramming (Special Issue on Parallel Embedded Real-Time Systems)*, 40:117–134, 1994.

[331] F. Tisato and F. DePaoli. On the duality between event-driven and time-drivern models. In *Proc. of the 13th IFAC DCCS*, Toulouse, France, 1995.

[332] Aviation Today. Parker selects TTTech for fly-by-wire system. Press release, July 2010.

[333] G. Torrisi, J. Notaro, G. Burlak, and M. Mirowski. Evolution and trends in automotive electrical distribution systems. In *Proc. of the IEEE Conference on Vehicle Power and Propulsion*, page 7, 7-9 2005.

[334] W. Townsley, A. Valencia, A. Rubens, G. Pall, G. Zorn, and B. Palter. Layer two tunneling protocol ”L2TP.” RFC 2661, Internet Engineering Task Force, August 1999.

[335] C. Trödhandl. Architectural requirements for TTP/A nodes. Master's thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[336] C.H. Tsai and C.W. Wu. Processor-programmable memory bist for bus-connected embedded memories. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 325 –330, 2001.

[337] TTChip. *TTP/C Controller C2: Controller Schedule (MEDL) Structure – Document Protocol Version 2.1*. Schönbrunner Strasse 7, A-1040 Vienna, Austria, September 2002.

[338] TTChip Entwicklungsges.m.b.H. *TTP/C Controller C2 Controller–Host Interface Description Document, Protocol Version 2.1*, November 2002.

[339] TTTech Computertechnik AG, Schönbrunner Strasse 7, A-1040 Vienna, Austria. *TTPPlan The Cluster Design Tool for the Time-Triggered Protocol TTP/C*, April 2002.

[340] TTTech Computertechnik AG. *Time-Triggered Protocol TTP/C, High-Level Specification Document, Document Number D-032-S-10-028, Protocol Version 1.1*, 2003.

[341] TTTech Computertechnik AG. *TTX-AUTOSAR FlexRay Stack User Manual, Document Number D-110-G-70-006, Document Edition 4.3.1*, 2009.

[342] TTTech Computertechnik AG. *Interface Control Document HS-COM Layer, Document Number D-115-G-10-005, Version 0.1.1*, 2010.

[343] TTTech Computertechnik AG. *TTP-Build User Manual, Document Number D-001-G-01-002, Manual Edition 8.1.4*, 2010.

[344] TTTech Computertechnik AG. *TTP-Plan User Manual, Document Number D-001-G-01-003, Manual Edition 8.1.2*, 2010.

[345] Honeywell Tuscon. Design, implementation, and verification of fault-tolerant modular aerospace controls, Honeywell ncc-1-377. http://shemesh.larc.nasa.gov/fm/talks/Honeywell–TTTech.ppt, accessed August 2010, April 2003. Aviation Safety Program Single Aircraft Accident Prevention. Coop. Agreement NCC-1-377.

[346] Vector Informatik GmbH. Product catalog ECU software, page 80-81: CAN embedded LIN communication. available at `www.vector.com`, 2010.

[347] P. Veríssimo, L. Rodrigues, and A. Casimiro. CesiumSpray: A precise and accurate global time service for large-scale systems. *Real-Time Systems*, 12(3):243–294, May 1997.

[348] D.D. Davidson and V.Y. Chiu. Fail-operational global time reference in a redundant synchronous data bus system. Patent Application US 2005/0102586 A1, Honeywell, May 12, 2005.

[349] C.J. Walter, M.M. Hugue, and N. Suri. *Advances in Ultra-Dependable Distributed Systems*. IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, January 1995.

[350] H.F. Wedde and W. Freund. Harmonious internal clock synchronization. In 12^{th} *Euromicro Conference on Real-Time Systems*, pages 175–182, Informatik III, Dortmund University, Dortmund, Germany, June 2000. IEEE Press.

[351] N. Weininger and D.D. Cofer. Modeling the ASCB-D synchronization algorithm with SPIN: A case study. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 93–112, Springer-Verlag, London, UK, 2000.

[352] J. Welch and L. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation (*formerly *Information and Control)*, 77(1):1–36, 1988.

[353] J. Widder. Booting clock synchronization in partially synchronous systems. In *DISC*, pages 121–135, 2003.

[354] A.T. Winfree. *The Geometry of Biological Time*. Springer Verlag, New York, 2001.

[355] B. Witwer. Developing the 777 airplane information management system (AIMS): a view from program start to one year of service. *Aerospace and Electronic Systems, IEEE Transactions on*, 33(2):637 –641, April 1997.

[356] www.softing.com. CAN, CANOpen, DeviceNet. Website, August 2010.

[357] J. Zhang. Improved on-line process fault diagnosis using stacked neural networks. In *Proc. of the International Conference on Control Applications*, pages 689 – 694, vol.2, 2002.

[358] W. Zheng, J. Chong, C. Pinello, S. Kanajan, and A. Sangiovanni-Vincentelli. Extensible and scalable time triggered scheduling. In *Fifth International Conference on Application of Concurrency to System Design, 2005. ACSD 2005*, pages 132–141, 2005.