INSTITUT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Code Generation and System Integration of Distributed Automotive Applications

Wolfgang Haberl

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 31.03.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.07.2011 angenommen.

Abstract

Over the past 40 years the automotive industry has experienced a huge shift from constructing mainly mechanical systems to designing embedded real-time systems. Current luxury class cars contain about 2000 individual functions, which are executed — oftentimes under hard real-time constraints — on a distributed platform of up to 70 computing nodes. Regarding their complexity and safety requirements, such automotive systems resemble their avionic counterparts, which evolved similarly. Software engineering for automotive systems, however, is essentially different from that employed in the avionic domain. While designers of avionic systems make extensive use of model-based engineering, the automotive industry still lacks a consensus what an ideal model-driven development should look like. With automotive manufacturers and their suppliers using various — often incompatible — tools, a comprehensive modeling of entire automotive systems is out of reach. This also applies to related model-based techniques like model checking, simulation, and code generation, as well as other typical benefits: lower development time, decreased development cost, and higher system quality.

This thesis presents an approach for the automated deployment of distributed automotive applications. The concept is based on the Component Language (COLA), which has been created for the design of automotive systems during a joint research project between BMW Group Research and Technology, and Technische Universität München. COLA facilitates software modeling throughout the entire development process by providing concepts for modeling requirements, functionality, and technical aspects of the system under design. With such a comprehensive model available, it is not only possible to derive executable code for applications, but also generate configuration data for the target platform. Thus, the generated system is ready for execution on a distributed platform without any manual integration required.

As an extension to the deployment approach, concepts for generating fault tolerance modes and for replaying runtime data in the COLA model simulator are presented in the thesis. Further, a concept for integration of COLA modeling and the Automotive Open System Architecture (AUTOSAR) is outlined. The viability of the described deployment approach is exemplified using two case studies.

Acknowledgements

First of all, I would like to thank my advisor Prof. Dr. Uwe Baumgarten for giving me the opportunity to work at his chair and for making it such an enjoyable time for me. He was always willing to listen and his advice and support were of enormous worth to me. Further I would like to thank my second advisor Prof. Dr. Johann Schlichter for accepting the judging of my thesis and for giving me valuable remarks.

Many thanks to all my colleagues at the Lehrstuhl für Betriebssysteme und Systemarchitektur and at the related chairs at TU München. I had a great time with all of them during productive meetings, inspiring discussions, and basic day-to-day work as well as during conference travel and leisure time. I would also like to thank the many students who helped me during implementation and testing of the tool prototypes and demonstrators for this thesis.

I am deeply grateful to my family and my friends who supported me over the years. They gave me the strength and motivation to achieve my aims and to overcome setbacks. Special thanks go to my uncle Bernhard Stimmel for proofreading my thesis.

Last but not least, I would like to thank my parents, my mother Irmgard Haberl and my father Walter Haberl. They were always there for me and backed my decisions. This thesis would not have been possible without all their love, encouragement, and support.

Contents

Chapter 1

Introduction

The thesis at hand describes an approach for the model-based generation of distributed embedded hard real-time systems, particularly those used in the automotive domain. These systems are characterized by their distributed nature, that is, they consist of a cluster of networked computing nodes, as well as their real-time critical properties, meaning severe hazards may arise from their failure. Consequently, the correct implementation of these highly complex systems is a must. The approach presented in this thesis aims at raising the quality of systems through the avoidance of programming and system integration faults. This improvement in quality can be achieved by employing models for system design and, as we will show, an automated transformation of those models into an executable system. In addition we will present a new debugging concept being integrated in the proposed system design process. Further we present an adaption of the concept to the *Automotive Open System Architecture* (AUTOSAR) standard as well as a methodology for the generation of fault tolerance mechanisms.

This chapter gives a short introduction into the evolution of automotive systems. Based on the insights described we will argue for the employment of *model-driven development* (MDD) in the domain. Subsequently, the limitations of MDD methods currently used in automotive software development are summarized and a problem statement is presented along with a possible solution. The chapter concludes with an overview of the organization of the remainder of the thesis.

1.1 Evolution of Automotive Systems

The electronic systems used in cars have significantly changed over the past decades. Initially electrically powered components, like servo motors, replaced their previous mechanic counterparts. The control of those components continually grew more and more complex by employing more capable control units, evolving from simple on/off switches to relays and later on to microcontrollers.

Consider for example a system which appears to be as simple as a windshield wiper. Initially operated manually in the early days of automobiles, the mechanics were soon driven by an electric motor. A simple switch was sufficient to start and stop operation of the wiper. The bidirectional motion was implemented using an arrangement of levers without altering the motor's rotational direction. Later the switch was replaced by a relais which was able to drive the motor in both directions and at different speeds, making the levers obsolete. In addition an intermittent mode for light rain was developed. Today wipers are controlled by one of the many *electronic control units* (ECU) contained in modern cars. These ECUs are built around microcontrollers which implement the desired behavior in software. The wiper motor is connected as an actuator and different sensors are used for governing the wiper's operation. By attaching a rain sensor the wiper may be turned on and off autonomously, and its speed can be adjusted to the amount of water on the windshield. Through governing the wiper's movement its speed can be kept constant, independently of the car's velocity and the occurring head wind. The driver's input is comprised as another source for input data altering parameters like the sensitivity of the rain sensor.

In actual automotive systems the number of ECUs employed is often related to the number of different user functions the car features. The evolution exemplified before led engineers to the practice of replacing each of the former subsystems by a distinct ECU and adding even more ECUs for newly invented functions. Thereby, today's automotive systems are huge networks of up to 80 ECUs, communicating via several bus systems [27, 26, 125]. This requires great effort regarding the cabling, increased weight, and difficulties with free space for the systems in an actual automobile. Of course, such a highly distributed system is extremely difficult to program and to integrate without severe faults.

Still the use of microcontrollers is very beneficial. It allows to implement differing behavior of an ECU for alternative applications which is of great benefit for its use in multiple product lines or new car models of a manufacturer. Besides that, the flexibility of changing a subsystems behavior simply by altering its software allows to mask deficiencies of the connected sensors and actuators. Finally, more complex, system-wide behaviors may be implemented if an ECU is able to interact with other ECUs.

To get the implementation of such a system under control demands for new development concepts which reduce the complexity for the developer. The need for such approaches is obvious as statistics show increasing car breakdown rates caused by electronics and software faults [42]. It is our belief that automotive systems can be highly improved regarding safety and reliability by employing MDD in combination with code generation and automatic configuration of the platform. If the model contains the necessary information, additional non-functional requirements like reduced overall power consumption or maximum hardware cost could also be satisfied. Unfortunately, an adequate MDD approach providing all necessary information for distributed code generation is missing, especially one providing a combination of soft- and hardware characteristics. We will propose a suitable approach in this thesis.

1.2 Model-Driven Development

In respect of the huge demands regarding safety of automotive systems, manual implementation can be considered error-prone. Even worse, the complexity of these embedded systems is increasing because of the growing amount of functionality realized by software, as has been presented by Ebert and Salecker [43]. In order to reduce the complexity developers have to deal with, especially in case of the distributed systems used in the automotive domain, MDD is a welcome facilitation. The use of models, which allows an abstraction of certain details of the target domain, makes the overall system easier to understand [48]. Coupling different types of models — or more precisely, different abstractions — results in a modeled system which still holds all relevant information. When combined with automatic code generation, MDD avoids most of the typical coding errors arising from manual programming. Adding a technique of platform configuration, turns system integration into an automated process, further reducing the risk of faults and making this time-consuming task redundant.

As a result the systems quality is further increased and the necessary development time is decreased. The time needed for development may be even more diminished due to a lower number of faults to debug, reducing the effort for changes and refactoring. These benefits result in a shorter time to market and reduced cost which is essential especially for the automotive domain. The mentioned advantages of MDD for embedded systems have been stated, amongst others, by Liggesmeyer and Trapp in [90] and Balasubramanian in [14].

Consequently, a multitude of MDD approaches and tools has been presented over the past years. In the automotive domain *MATLAB/Simulink*[1] and *ASCET-SD*[2] are the best-known commercial tools for designing and generating functional code. The *Unified Modeling Language* (UML) [114] and *EAST-ADL* [40] in contrast, focus on modeling the architecture of the overall system and its behavior at a rather abstract level. The lack of any formal semantics for those modeling languages makes code generation at best tool specific, if not impossible, when the resulting code shall be used unaltered in an actual system. Without defined semantics the models can neither be verified by model checkers, nor can their transformation into code be guaranteed to be consistent across different implementations. Rather, rough drafts as a base for manual coding can be derived. We will illustrate the shortcomings of those tools in Section 4.5.

1.3 Limitations of Current Software Development Approaches

As stated by Lee [88] embedded systems development failed to catch the interest of computer scientists in the early days of such systems. Over time embedded systems grew more and more complex as their functionality and safety-relevance increased. Today, automotive and avionic systems require huge effort for their implementation and integration. Hence, new development concepts which lead to less faults and decreased time-to-market are highly welcome.

In the course of dealing with this issue, MDD is seen as a viable solution to increase quality of the resulting systems. Several approaches and tools have been created for this purpose. A common shortcoming of these concepts is that they all focus on certain aspects of systems development rather than providing support throughout the entire process. Consider for example the afore mentioned modeling tools MATLAB/Simulink and ASCET-SD which are today used by *original equipment manufacturers* (OEM) and their suppliers in the automotive sector. These tools focus on designing and optionally generating the functional code for an ECU. What they do not provide is a means of specifying a distributed system's software architecture, let alone its hardware architecture. Without these informations they cannot enable code generation for distributed applications without manual integration effort. Consequently, these tools are rather used for designing single functions or, at most, the code for a single ECU. Systems integration remains a subsequent, manual step.

At the other end of the spectrum are concepts for the design of architectural aspects of an automotive system like EAST-ADL or UML, that allow a high-level description of the distribution aspects of software and hardware. At the same

[1]http://www.mathworks.com
[2]http://www.etas.com

time they fall short when considering behavioral modeling at a detail-level suited for code generation. Rather they allow to optionally integrate other modeling languages for functional modeling like MATLAB/Simulink. But this combination of different modeling languages bears the inherent risk of incompatibilities or a loss of information at the contact points. And if in turn the functional code is implemented manually rather than generated automatically, the risk of coding errors is huge. To avoid basic coding errors, automatic code generation is seen as a must by scientists familiar with the safety-critical domain, as stated by Sangiovanni-Vincentelli and Di Natale [116].

The analysis of these limitations together with the specifics of embedded hard real-time systems and the available MDD tools enables us to derive the following problem statement.

1.4 Problem Statement

In order to tackle the complexity and safety requirements of automotive systems, model-driven development is seen as a possible solution. Yet the existing approaches and tools focus solely on few of the aspects of the target system. We can distinguish two classes of tools: the first class of tools currently used allows for functional modeling and code generation. They are aimed at designing single software functions, ignoring distribution problems. However, the second class of tools, which are intended for architecture modeling of distributed systems, does not enable developers to carry out a functional specification of the system which is detailed enough to generate code and configure the distributed system. Thus, there is not a single integrated MDD approach, ranging from requirements specification over architecture modeling down to functional and technical specifications needed for code generation and configuration of a distributed platform [90].

With the lack of an integrated modeling approach, code generation and automated configuration of distributed embedded hard real-time platforms is unfortunately out of reach. Since current tools impede an automation of these steps through a lack of comprehensive system information, manual coding and system integration are required, which are some of the most error-prone tasks during embedded systems development. As a result, systems are often faulty and require a huge effort for bugfixing to conform to the necessary safety requirements.

Over the recent years an approach supporting all steps from requirements specification to functional and platform modeling has been developed at TU München [55]. The approach is built around the Component Language (COLA) [86], which provides modeling artifacts along the entire development process. A COLA model comprehends all information relevant for the automatic generation of an executable system. Thereby, the COLA development approach implements the demanded integrated approach and solves the previously mentioned problems.

1.5 The Solution — Model-Based Code Generation and System Integration

The deployment concept presented in this thesis aims at generating distributed application code and platform configuration data. The term *deployment* is used in various contexts within the field of computer sciences. Oftentimes, it is applied to describe the job of rolling out a certain piece of software onto numerous identical target machines. In this thesis, deployment is used in a slightly different sense:

Definition 1.1. *In the thesis at hand the term* ***deployment*** *indicates the transformation of a modeled system into an executable target system. This transformation includes generation of functional code as well as configuration data for a distributed target platform.*

Platform configuration includes processor and communication schedules along with a communication matrix for an automotive platform. As a result the modeled applications, for which code has been generated, do not need to be integrated manually. Rather the integration is already achieved by the code generation tools and the temporal behavior of the system is fixed. In order to provide this determinism a future automotive target platform is necessary which features several key characteristics. In this section we will give an outline of the code generation solution along with our vision for a suitable future target platform. In the following, we will give the rationale why this platform is a realistic vision.

Solution approach. The code generation concepts presented in this thesis are part of a project having been carried out in the context of the Car@TUM cooperation. This cooperation is a framework for scientific projects of Technische Universität München and BMW Group. The main goal of the project was to find a solution for the model-driven development of automotive systems not being limited to certain steps along the development process, like algorithm design or task distribution on the target platform, but covering all necessary steps. To this end the Component Language has been invented providing modeling artifacts for requirements specification as well as functional design and technical views of the system under development. Together with the language a prototype tool-chain for system development support was created. This tool-chain implements an integrated MDD approach, making the use of adapters for heterogeneous stand-alone tools and manual data migration — as is state-of-the-art nowadays [63] — redundant.

COLA features formal semantics which enable the use of model checking during development. While not all properties of a model can be checked due to computational complexity, the resulting model is already of higher quality. To retain this quality down to executable code on a distributed platform, the code generation tools described in this thesis are used. The application code generator uses the functional system design given in COLA for its translation into C Code. The allocation of modeled tasks to computing nodes of the hardware platform is specified in the technical view of the COLA model. Furthermore the COLA model contains information about the hardware platform including processing speeds, memory capacities and bus systems. This information is used to calculate valid schedules and a communication matrix. The schedules and the communication configuration files are used by the target platform during execution of the generated system. We will show that the information contained in a COLA model can be employed to further enhance the system quality by debugging design faults at model level and by generating fault-tolerant systems. We have given a detailed introduction to the COLA development process in [55]. The different steps of this process will be recapitulated in this thesis.

To ensure correct behavior of the target system, a custom architecture is envisioned for the platform consisting of hardware, operating system, and middleware. This vision is guided by hard real-time constraints and experiences from the avionic domain, which is similar to the automotive domain regarding its safety requirements. Next we will sketch our vision for a future automotive platform which is suitable to back the described system generation concepts.

Target scenario. The deployment described here is aimed at the typical control loop applications used in automobiles. This includes safety-critical functions for driver assistance, passenger safety, comfort electronics, and similar functions. The part where this concept is not significant for are entertainment functions like radio, television, navigation, etc., which are in modern luxury cars provided by an ECU referred to as *head unit*. While the head unit is typically a single processing node, control loop functions are distributed over tens of nodes connected by several bus systems and types, as has been mentioned before.

In order to achieve the timing requirements necessary for the targeted control loop applications, we rely on time-triggered scheduling rather than using a priority-backed event-triggered approach. The benefit of time-triggered scheduling lies in its deterministic behavior. If computing times for all tasks as well as their deadlines and processing power of the target platform are known, valid schedules meeting all timing requirements can be calculated offline. To achieve a time-triggered execution for distributed tasks independent single-ECU schedules are not sufficient. Rather a system wide scheduling plan has to be used. This plan contains starting points for all tasks, along with communication slots in between. The execution of this plan demands a global clock on all nodes of the system. In combination with a time-triggered bus protocol such a platform yields guaranteed timing behavior.

Hence, our platform concept envisions a middleware which is based on a time-triggered bus and provides access to a globally synchronized clock. In addition the middleware offers an abstraction from specific bus systems and addressing schemes to the applications. This property not only simplifies code generation, but also facilitates the reuse of software [63, 103]. The middleware's configuration is achieved by a configuration tool that maps data exchange between distributed software components of the COLA model to middleware communication. The middleware is intended to be configurable accordingly at system start-up. The use of a standardized middleware API and logical addressing allows for the generation of code, even before an allocation decision has been made. In addition a relocation of tasks from one ECU to another is facilitated, possibly even at runtime. The middleware shall also be used by applications for hardware access. By avoiding direct calls from the application to the hardware, remote use of hardware connected to a different ECU is feasible.

The named characteristics are similar to those of the TTA proposed by Kopetz in [80]. We will explain the differences between the TTA and the approach proposed here in Chapter 2. The following section explains why the outlined platform is a realistic vision for future automotive systems.

Scenario justification. Over the recent years automotive software grew more and more complex. Today up to 90 percent of innovation in automobiles is due to software or at least supported by it, as shown by Broy in [26]. This results in an increased overall number of nodes as well as the use of more powerful microcontrollers in the individual nodes.

The avionic domain has been faced even earlier by a similar growth of safety-critical software systems. In order to reduce the number of individual nodes, aiming at less weight, cabling, and required space, the concept of *integrated modular avionics* (IMA) [5, 109] has been created. IMA proposes the use of a small number of powerful multi-purpose computing nodes, each replacing a large number of the smaller legacy nodes. By employing time-triggered task execution and communication according to the ARINC methodology [20, 33, 36, 110], timely execution of the integrated safety-critical applications can still be ensured, despite a lower overall number of processors.

The transfer of these avionic concepts to the automotive domain fits the target scenario described before very well. The evolution towards a homogenous automotive platform, consisting of a number of powerful computing nodes seems immanent, just as it happened with avionic platforms. This requires the same time-triggered mechanisms that are already used in airplanes. Otherwise a prediction of the systems' timing behavior would be impossible.

The use of fewer processors in such a platform leads to a better overall utilization of the processing power available. Lots of the microprocessors used in actual cars are executing NOPs, that is, they have no operations to execute and burn processing cycles most of the time. Consider for example a power window controller that is rather rarely activated. Despite being not required this ECU is active all the time, consuming energy when the ignition is turned on.

As for the middleware proposed in this thesis, a similar hardware abstraction layer is already emerging for automotive systems. Looking at the interfaces described in the AUTOSAR standard, a unified access model for bus communication and hardware interfacing is proposed. The middleware described in this thesis extends the AUTOSAR concept towards flexibility at runtime and additional middleware services, as we will show.

1.6 Outline of the Thesis

The remainder of this thesis is structured as follows:

Chapter 2 — Foundations of Embedded Real-Time Systems. In Chapter 2 we give a more detailed introduction into the field of embedded real-time systems. The reader is given a deeper insight into the specific challenges during the design and implementation of safety critical systems being subject to real-time constraints and distributed topology. The automotive systems targeted in the thesis are clearly within that field.

Chapter 3 — Specifics of the Target Domain. Subsequently, in Chapter 3 the complexity, safety, and cost requirements of automotive systems are discussed. Furthermore, a summary of current automotive software development concepts is given. The chapter concludes with an outlook on future trends for automotive systems.

Chapter 4 — The COLA Approach. Chapter 4 begins with a description of our vision for model-driven development of automotive systems. Subsequently it provides an introduction to synchronous dataflow languages. Next, the COLA modeling language and its concepts are introduced, as well as the generic tool-chain built around it. As for the tool-chain, the focus of this chapter is more on the modeling and verification tools, since the code generators used by the deployment concept presented in this thesis are detailed in Chapter 5. Further, a summary of related work targeted at MDD of automotive applications is given.

Chapter 5 — Fully Automatic Deployment. The main contributions of this thesis are presented in Chapter 5. The deployment concept presented here builds on the modeling concepts of the COLA approach introduced in the chapter before. Using the deployment tools, distributed applications can be generated from a COLA model, rendering redundant any need for manual coding or integration. The generated artifacts include appropriate configuration data for the distributed execution platform. The chapter starts with an overview of the deployment process and the requirements for a suitable target platform. Afterwards, the custom middleware is introduced, before we outline the different deployment tools for transforming the model into an executable system.

Chapter 6 — Extensions of the Deployment Concept. Because COLA provides an integrated model covering the entire automotive system, additional benefits besides the basic code generation can be shown. One additional concept which has been accomplished is a mature debugging approach, backing up the code generation with a reverse mapping of runtime data to model elements. This facilitates debugging at model level, without any manual coding. Another concept enables the generation of fault tolerance modes, enabling the implementation of software redundancy in an automated manner.

The concepts presented in this thesis require a specific target platform to take full advantage of the COLA approach. The automotive industry already began to migrate towards a different platform standard named AUTOSAR, which is supported by many well-known automotive OEMs like BMW, Volkswagen, Daimler, Ford, Toyota, as well as a major number of their suppliers. While some of the ideas presented in this thesis cannot be carried out using an AUTOSAR platform, basic code generation for AUTOSAR from COLA models is still possible. We will explain the key differences between AUTOSAR and the original platform concept along with possible modifications of the deployment tools in this chapter.

Chapter 7 — Evaluation of Concepts. In order to show the viability of the deployment concept, two case studies have been completed. These case studies are presented in Chapter 7. The first case study illustrates the generation of code from a COLA model for an undistributed platform. The second case study shows a distributed platform, thus demonstrating the full functionality of the deployment approach. Besides testing the deployment tool prototypes, the second case study was also used to exemplify the model-level debugging approach introduced in Chapter 6.1.

Chapter 8 — Conclusions. In Chapter 8 we will summarize the achievements of the presented approach and discuss its integration into the current automotive software development process. Finally, we give some future prospects for software development in the automotive domain.

Chapter 2

Foundations of Embedded Real-Time Systems

The software development concept presented in this thesis is targeted at automotive systems, which are embedded real-time systems from a more abstract point of view. Real-time systems are typically operating under stringent timing constraints. The moment in time the system sends a results is as important as the result itself for correct behavior. As these systems are not visible as computers in an automobile, but rather interact with their environment using sensors and actuators, they are called embedded systems. In addition, automotive systems are typically distributed which refers to their structure as an interconnected network of computing nodes, exchanging their respective current states. We will summarize the key aspects of real-time and embedded systems in Sections 2.1 and 2.2, respectively, to give a better understanding of the general conditions for the implementation of automotive systems. In Section 2.3, we want to give a short overview of the actual practice of embedded real-time systems development. Finally, in Section 2.4 we will introduce the time-triggered architecture, a well accepted standard for the implementation of highly safety critical real-time systems.

2.1 Real-Time Systems

As described, for example, by Koptez [79] a *real-time computer system* is characterized by the fact that the correctness of its operation is not only dependent on the logical results, but also on the point in time at which these results are produced. The real-time computer system is the central part of each *real-time system*, which additionally compromises an *operator* and a *controlled object*. As depicted in Figure 2.1 the operator controls the operation of the real-time computer system using a *man machine interface*. The real-time computer system interacts with the controlled object accordingly via the *instrumentation interface*. For some real-time systems the operator is optional. In that case, the real-time computer system is programmed with a fixed parameter set how to modify the controlled object, according to the current sensor values.

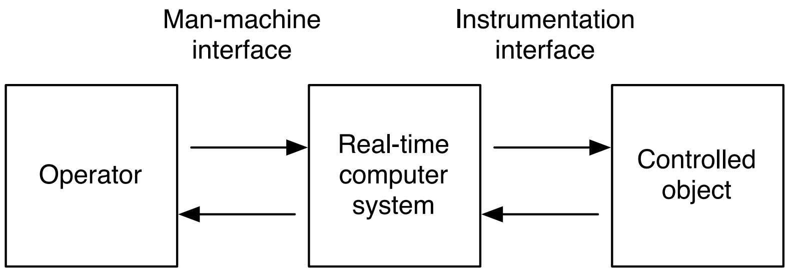


Figure 2.1: Real-time system (cf. [79])

The temporal requirements for real-time systems arise from their use for *control loop applications*. The *control loop* is established between the real-time computer system and its environment, consisting of operator and controlled object. The computer system reads sensor data which correspond to the actual state of the controlled object as well as user input. Using these data the computer system is able to calculate possible deviations of the object's actual state from the intended optimal state. Based on the result the computer system triggers actuators to influence the state of the controlled object. The result of this modification can subsequently be seen in changing sensor data. So the system reacts continuously and instantly to its changing environment. Hence, control loop systems are also defined as *reactive systems*. This term has been introduced by Harel in [65].

Figure 2.2 shows the control loop for an automotive *anti-lock braking system* (ABS). As soon as the operator pushes the brake pedal, the brake is slowing down the car's wheels. The anti-lock brake controller, which is implemented by a real-time computer system, monitors the revolutions per minute (RPM) for each wheel using wheel speed sensors. If it detects a sudden change in the RPM, a change to zero in particular, the system assumes that the wheel blocked due to braking intervention. The anti-lock brake controller reacts to this situation by opening the hydraulic valve for the respective wheel until its RPM is detected to be within the

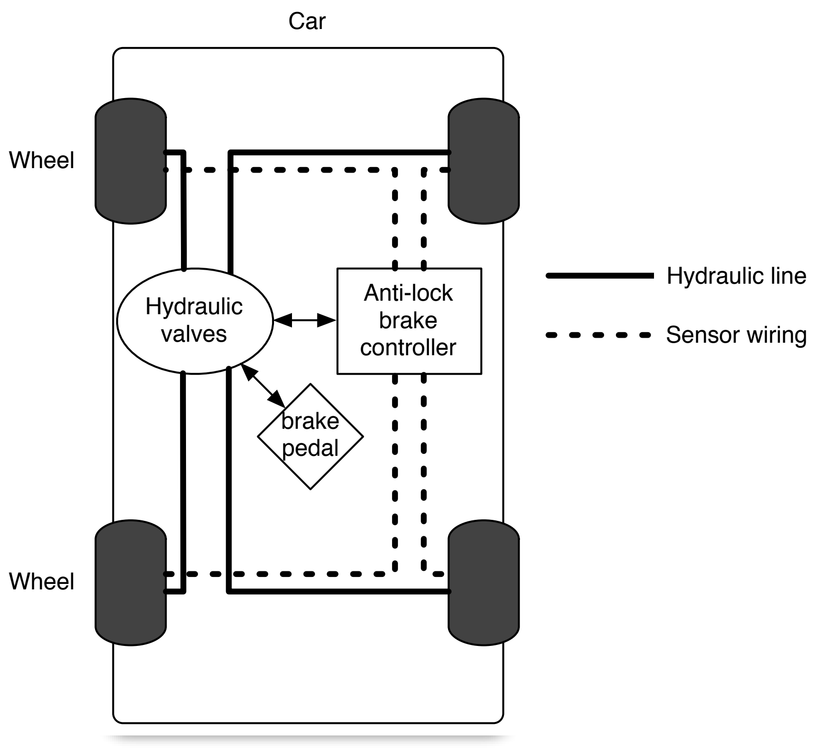


Figure 2.2: Control loop of an anti-lock braking system

given boundary. Then the valve is closed, providing the wheel with full hydraulic pressure again.

2.1.1 Soft and Hard Real-Time

Real-time systems can be categorized into *soft* and *hard real-time* systems. In a soft real-time system the occasionally delay or omission of a result leads to a degradation of the intended functionality. In a hard real-time system late or missing results compromise its safety, which might lead to huge warranty costs, human injury, or even casualties. An example for a soft real-time system would be a home theatre computer, where delayed sound playback during a movie might annoy the user. But this problem is surely not fatal for the user. In contrast a car is considered a hard real-time system as the failure of one of the safety-critical ECUs could entail an accident. The afore described example of an anti-locking braking system falls clearly in the latter category.

Guaranteeing the timely operation of a hard real-time system requires predictable behavior even under peak-load. This demands a more conservative allocation of computing resources, which can guarantee timely execution of all tasks. *Worst-case execution times* (WCET) of all tasks have to be known to facilitate the analysis of schedulability of the task set. Further, all delays in the system caused by jitter, fetching data, and operating system overhead, have to be restricted to a known maximum, as Silberschatz et al. point out in [117]. For the same reason advanced operating system features like virtual memory are not employed, since they tend to provoke unknown overhead.

For a soft real-time system, it is acceptable to provide degraded service in case of peak-load. Hence, the requirements towards the timeliness of such a system are less strict than for a hard real-time system. Soft real-time systems typically give critical tasks highest priority until they finished their execution. The exact duration of the tasks' execution is not necessarily known a priori. This approach can lead to occasional deadline violations, which is — within limits — acceptable for soft real-time systems.

Hard real-time systems in contrast may not miss any deadline. Consequently, the scheduling algorithms used for hard real-time systems have to guarantee the timely execution of all tasks. To this end offline calculation of schedules is oftentimes employed [132]. Priorities and starting times for the tasks are carefully assigned to avoid deadline misses due to priority inversion or delayed process execution.

The majority of systems in an automobile are hard real-time systems. Since these systems provide essential functions needed for the safe operation of the vehicle, they must not fail. We will thus focus on this kind of real-time systems in the remainder of this thesis.

2.1.2 Real-Time Communication

Real-time systems are often of distributed nature. A distributed real-time system architecture consists of a set of nodes and a communication network between these nodes. Thus, the system's functionality is not necessarily implemented on a single node, but may be attained by an arbitrary number of cooperating nodes in the system. From a functional point of view there is no difference whether the computation is distributed over several nodes, or executed on a single node. The distributed solution, however, yields several advantages:

Architectural considerations: It may be advantageous to execute a function on dedicated hardware. This approach allows the employment of hardware components, for example a digital signal processor, which are optimized for the respective function. The resulting node is able to calculate results faster or more efficient. If several nodes with identical hardware architecture can be used in the overall system, the opportunity of mass production might also reduce the cost per node.

Other constraints for the system's architecture may originate from the historical evolution of the system. Considering an automotive system, the functionality of subsystems has been largely improved by replacing simple circuitry with micro controllers. The resulting system is a large set of small subsystems, which are also referred to as *nodes* of the system. Often it is easier to situate the simpler, and hence smaller, subsystems in the installation space available, than larger pieces of hardware. At the same time, the cabling might be reduced if each subsystem's physical location is close to the sensors and actuators it uses, compared to a star topology in a centralized system.

Further, from a system architect's point of view, the abstraction provided by the distributed nodes eases the design of the overall system. As explained by Kopetz [79], each node is an autonomous real-time system with essential functional and temporal properties. The exact implementation of the node is hidden beneath its interface.

Fault containment: Hiding a node's internal state is also advantageous in case of failure of a node. Since each node is implemented as an autonomous subsystem, it can be prevented from influencing the rest of the system if it fails. The communication system is liable for providing a mechanism which avoids the propagation of erroneous values to other nodes of the system. This approach largely improves the reliability, and thus safety, of the overall system.

Extensibility: With all subsystems providing a common interface, easy extensibility of the real-time system is achieved. If the communication system provides the necessary resources, it is sufficient from an architect's point of view to add further nodes to the system by connecting them to the communication system. Of course this approach requires the subsystems to be *composable*, that is, their integration into the overall system via the communication mechanism must not invalidate the timeliness property achieved by the system.

Parallelism: The availability of a larger number of processors and memory, compared to a single node solution, enables the parallel execution of tasks. This is mainly of interest for large scale systems, where several independent tasks are ready for computation at the same time. Parallelization of the system not only improves processing speeds, but also lowers overall system cost. As presented by Patterson and Hennessy in [106], the approximate cost of a chip is proportional to the third power of the die area: $Cost = K * (Die\ area)^3$ (with K being the cost factor for process and production technology employed). Hence, adding more processors to the system is, at least in the long term, a more cost effective method to increase performance than employing more capable processors with a larger number of transistors. The increase in cost for adding processors is only linear, as depicted in Figure 2.3. Still, the ideal sizing of the employed processors is dependent on the respective executed tasks.

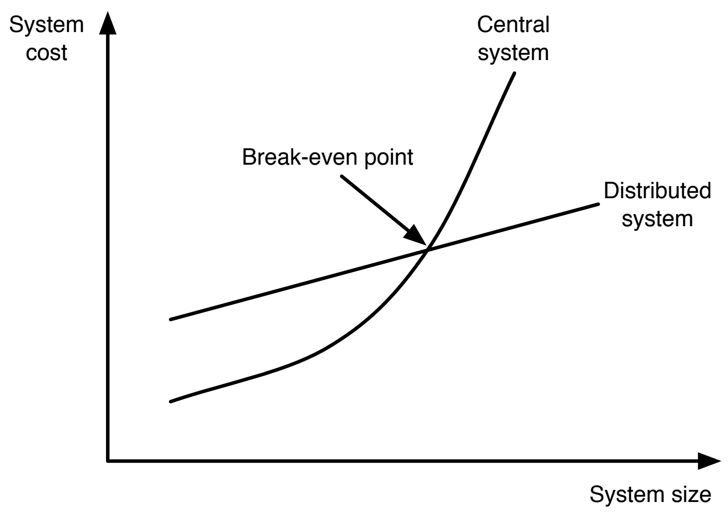


Figure 2.3: Cost growth of centralized versus distributed architectures (cf. [79])

Event and state messages. The nodes contained in a distributed real-time system exchange their view of the system and its environment by means of messages. This enables the nodes to acquire other input data than just those produced by their directly connected sensors or the locally executed tasks. Regarding message semantics, *event messages* and *state messages* are distinguished. Event messages are triggered by an event occurring in the sending node, which is instantly forwarded to a receiving node. Typically, the message is only sent once and it is necessary for the receiver to get and process the message. Otherwise, the system state as seen by the sender is different from the state assumed by the receiver. Guaranteeing the reception and interpretation not only requires a reliable communication system, but also a synchronous communication between sender and receiver. If the receiver would not process the messages instantly, its message buffer might overflow in case of a large number of messages — sent by either one or several nodes. Event-message semantics are used by most non real-time systems.

State messages, in contrast, provide cyclic updates of a node's state. They are sent at a defined frequency and contain the current state of the sending node, as far as it is important for the other nodes in the system. State messages are automatically buffered by the communication controller of the receiving node. If a receiving node does not process one of these messages, a subsequent update simply overwrites the buffered data. Since the applications in a state message based system are aware of possibly missed updates, this loss of data does not influence the system's operation in a negative way.

Event- and time-triggered communication systems. In an *event-triggered communication system* each communication partner may start to send an event message at a random point in time. Hence, the timing of the communication is controlled by the host computers rather than the communication system. The need for processing messages momentarily requires the application processor to interprete this data as soon as a incoming message is detected. Thus, the execution of other tasks may be delayed in an unpredictable manner depending on the number of messages. The receiver may also be overloaded, if a larger number of sending nodes decides to send messages to it at the same point in time. Furthermore, if a shared medium is used the simultaneous access of sending nodes to the communication system may cause unpredictable delays. *Ethernet* is a prominent example of a bus protocol using a shared medium and random access technique. Other protocols like *Token Ring* or the *Controller Area Network* (CAN) [121] make use of tokens or message priorities, respectively, to control access to the medium. Still, they rely on the application processor to take care of the timely transmission and reception of messages.

Time-triggered communication systems make use of dedicated communication controllers which take care of the timing for sending and receiving messages. State messages are transmitted cyclically at predefined points in time. The exact timing is transparent to the sending and receiving applications. They only interface with the communication system for sharing data. The schedule for accessing the communication medium is stored in every communication controller, assigning dedicated sending slots to each sender. This approach avoids message collisions at the medium level. Since the communication schedules are defined at design time of the system, the temporal behavior at the communication interface is fixed and can be tested for each single node. The system is now predictable in contrast to a system based on an event-triggered communication system, whose behavior differs with the number of messages sent [77, 78]. Examples for time-triggered (tt) bus protocols are *FlexRay* [96] and the *Media Oriented Systems Transport* (MOST)[1], which are already employed in actual automotive systems. Another prominent example is the *Time-Triggered Protocol* (TTP), which is used in the *Time-Triggered Architecture* (TTA). We will give an introduction to the TTA in Section 2.4. All these protocols assign time slots for communication to regulate access to the medium. Further, the protocols are implemented in form of bus controllers, taking care of the necessary message timing.

2.1.3 Clock Synchronization

In a system whose quality is dependent on the production of results within given time bounds, a common view onto the current time is advantageous, if not even fundamental. The time may serve as an input to the executed applications, as well as for synchronization of the distributed execution of these applications. To this end, all nodes of a system should maintain a matching reference time, which is

[1]http://www.mostcooperation.com

called *global time*. The global time is — logically — kept by a *global clock*. Since typically not all nodes have direct access to a suitable reference clock, they have to employ their respective local timers for keeping track of time. These timers are, however, not able to provide exact synchronous time since they will always differ by a tiny bit from each other due to hardware restrictions. Hence, a means of clock synchronization is necessary. This synchronization makes sure that the deviation in each node's view of the global time is always within an upper and lower bound.

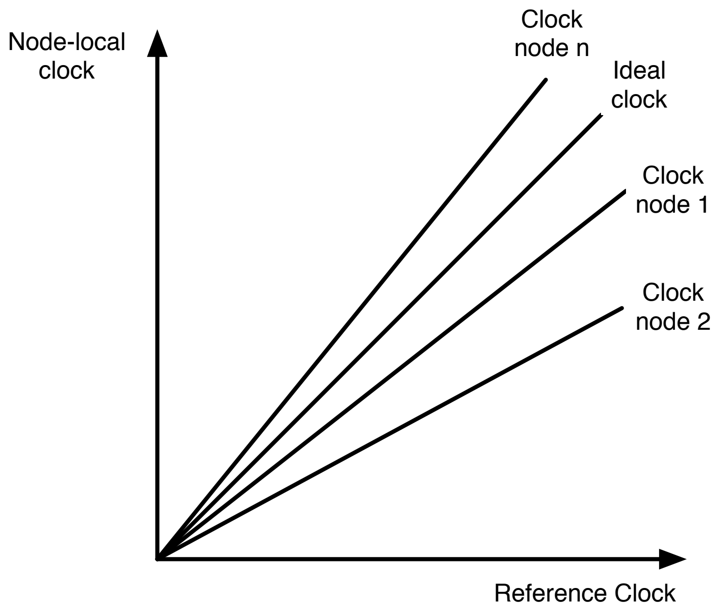


Figure 2.4: Drift between ideal clock and local clocks of nodes

Figure 2.4 depicts, how the time measured by a local clock deviates more and more from the reference time during system execution. If the local clocks would measure time perfectly, they would match the *ideal clock* shown in the figure. The deviation between ideal clock and a local clock is called *drift*. The amount of drift over a certain period is denoted as the *drift rate*. To keep the drift within limits requires cyclic synchronization of all clocks in the system. The period between synchronizations has to be chosen based on the desired maximum drift and the drift rate of the timers employed in the system. Figure 2.5 shows, how the local drift of each node can be kept within upper and lower bounds by synchronizing the respective clocks cyclically.

Often no external reference clock providing the needed precision is available. Therefore, a local synchronization protocol has to be used. Various protocols for such an internal clock synchronization have been proposed [83, 113]. These protocols can be categorized into central master and distributed solutions. For a central master algorithm one of the nodes of the system is declared to be the master node for clock synchronization, all other nodes are slaves in this protocol. The master's

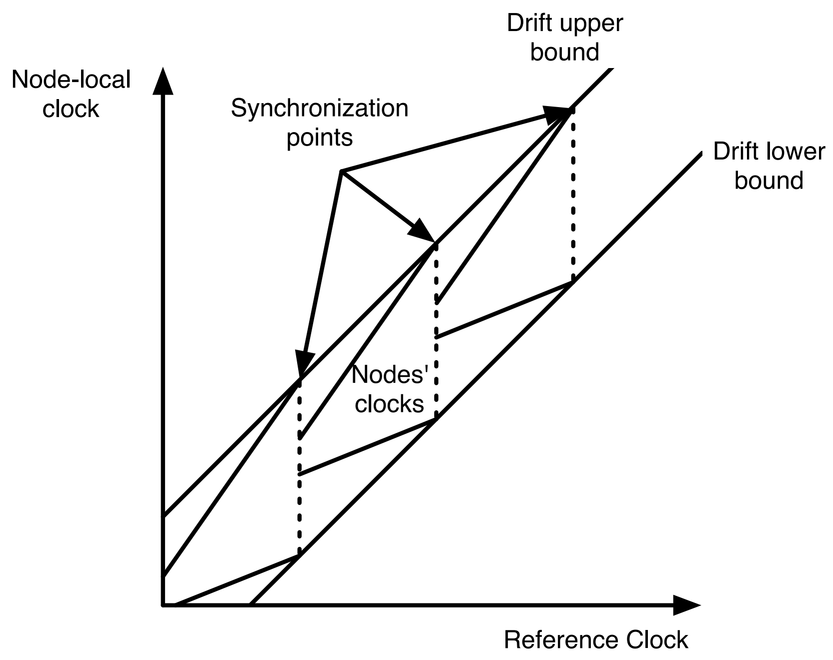


Figure 2.5: Bounded drift by synchronizing local clocks (cf. [79])

local time is assumed to be correct and therefore used for clock synchronization. To this end the master sends its local time as the system's reference time to all slaves, which update their respective local clocks, accordingly. Distributed synchronization is achieved by exchanging information about the respective local time between all nodes of the system. Each node uses the received information to calculate its own drift and correct is local timer accordingly.

For all synchronization protocols, message transmission times, jitter, delay, etc. have to be applied to the time value contained in the synchronization message. Faulty timers which exhibit a larger drift rate than specified, or even change their value randomly, have to be taken into account. Finally, a possibility for applying corrections to the local timer value without influencing the applications currently executed on the node has to be provided. For the implementation of these requirements, please consider the algorithms quoted above.

2.1.4 Hardware Devices

Distributed real-time systems interface a lot of sensors and actuators in order to interact with their environment. These devices are either directly connected to a node by dedicated cabling, or may be interfaced by a *field bus*. Field busses are rather slow, but very robust communication mechanisms, relying often on unshielded two-wire cables, keeping the costs low. Their performance is sufficient for transmitting the small datagrams used sensors and actuators in time. By offering the possibility to connect several devices to them field busses further reduce the amount of cables needed and thus lower cost and weight of the system.

For interacting with a node of the system, sensors and actuators use two different mechanisms:

Interrupt-driven: In an interrupt-driven system, communication with a device is initiated by the device itself, not the node. The device uses an interrupt line to signal the availability of data to the respective node. As soon as the node receives the interrupt, it pauses its current task and executes a handling routine for the interrupt in question.

The approach ensures minimum latency, since the node reacts instantly to the device's signal. This concept is the standard mechanism in desktop systems. However, the approach yields some problems for a real-time system. First, if more than one device is connected to the node, an order has to be specified on the interrupts. If more than one interrupt is generated at the same time, the higher priority interrupt is served first and the lower priority interrupt accordingly delayed. As a result, minmal latency can only be guaranteed for the highest level interrupt in any node. Second, the processing of the interrupt handling routine delays execution of the currently active task. In extreme cases, where lots of interrupts arrive close to each other, this might cause an application task to miss its deadline.

Polling: In a system based on polling, devices are not allowed to interrupt the application processor of a node. Rather, the node itself keeps temporal control and devices may deliver data when, and only when, the node asks them to do so. This requires the devices to contain a dedicated microcontroller able to communicate with the application processor of the connected node. This dedicated microcontroller is used to store values temporarily during the polling interval, that is, between the points in time at which the application processor is requesting data. Due to the use of a dedicated microcontroller such devices are also called *intelligent devices*. As a side effect these devices are oftentimes connected to a bus system, which enables their use by several application processors and reduces the amount of cabling necessary. Due to these advantages, intelligent devices are becoming more and more common [101].

The polling solution results in higher latency times than an interrupt-driven one, because the data acquired by a device are not delivered to the application processor requesting them. Thus the worst-case latency is expanded by the duration of the polling interval. In addition, a polling approach might result in some overhead because the polling interval is not necessarily related to the availability of data. Therefore, a lot of polling requests might return old or even no data. There again, the predictability of a polling solution facilitates guaranteed maximum reaction times, independently of the number of interrupts. And — even more important — the execution times of applications on the respective node are not influenced, as described for the interrupt-driven case.

Overall, the polling approach brings about somewhat more overhead — especially in case of sparse data — and increased latency times. The latter problem can be diminished by choosing an appropriate polling interval. On the plus side, the concept of polling results in a higher predictability of the system, which is highly welcome for hard real-time systems.

2.1.5 Real-Time Scheduling

Scheduling tasks — or processes, from an operating system's point of view — in a real-time system is subject to the respective deadlines required for these tasks. As indicated in Figure 2.6, the execution of a task is only beneficial if it delivers a result after its starting time and within its deadline. In the case of a safety-critical systems, shown in Figure 2.6, results generated before the start time or after the deadline might even cause damage and negatively impact the system's safety requirements.

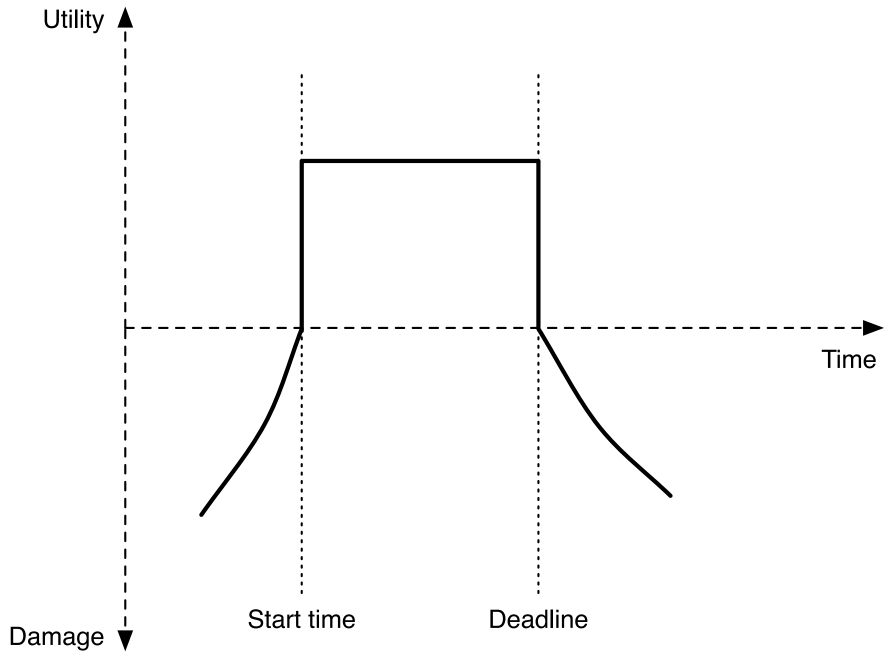


Figure 2.6: A safety-critical real-time system (cf. [31])

Processes. Processes can be categorized into two distinct forms: *periodic* and *aperiodic processes*. Periodic processes are executed on a regular basis and can be characterized according to their *period*, *deadline* and *worst-case execution time*. Aperiodic processes, however, have unknown starting points being related to some event. Since lots of such events can occur at the same instant — called a *burst* — a temporary overload of the executing processor cannot be averted. Consequently, as explained by Burns in [31], aperiodic processes cannot have hard deadlines. A worst-case calculation could only be facilitated by specifying minimum periods for each event.

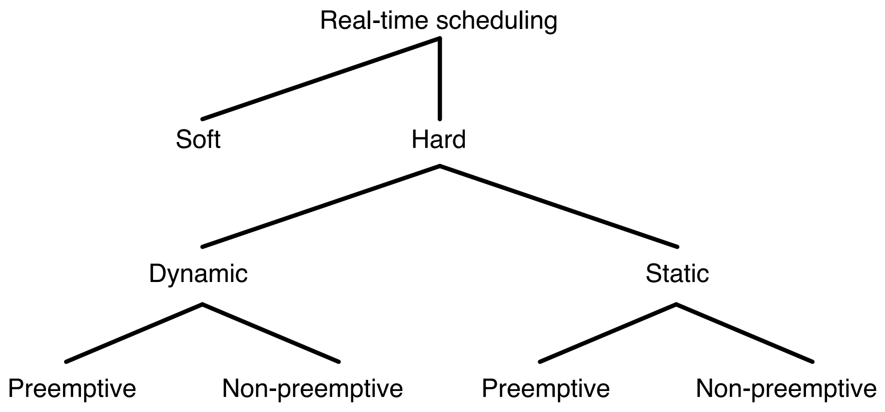


Figure 2.7: Taxonomy of real-time scheduling algorithms (cf. [79])

Scheduling algorithms. Figure 2.7 gives an categorization of scheduling algorithms for real-time systems. As shown in the figure, a basic distinction is made into algorithms for soft and hard real-time systems. As explained before, we are focusing on hard real-time in this thesis. The according algorithms are either of *static* or *dynamic* nature. Static algorithms are being computed offline, that is, before the target system is executed. The resulting pre-set schedules are stored in form of tables, based on the a priori knowledge about all processes. During execution of the target system, the scheduler acts simply as a dispatcher, assigning the processor according to the pre-calculated table. Dynamic algorithms, in contrast, select the active process based on the current set of ready processes and their respective priorities. Hence, every process has to be assigned a priority before execution. Static as well as dynamic scheduling algorithms can be *preemptive* or *non-preemptive*. Accordingly, a running process may either be paused by another higher priority process becoming ready, or not.

As explained by Burns in [31], dynamic algorithms are rather suited for soft real-time systems, due to their typical usage with aperiodic tasks triggered by external events. In a hard real-time system it is undesirable for unpredicted events to occur. However, using solely periodic tasks with static scheduling may result in low processor utilization, if these tasks frequently need less execution time than their specified WCET. Still, the resulting system is guaranteed to show predictable behavior. The same can only be achieved for dynamic scheduling of aperiodic tasks, if a worst-case assumption about their maximum number of invocations is made. The reservation of an appropriate amount of processing time would cause identical processor utilization as a periodic invocation of the same task. Therefore, the static scheduling approach is mandatory for hard real-time tasks. As stated by Xu and Parnas in [132], pre-runtime scheduling is often the only practical solution to achieve a predictable complex real-time system.

2.2 Embedded Systems

Embedded systems are ubiquitous today. More than 90 percent of all processors produced are already used in embedded systems [26, 74, 43]. In contrast to other computer systems, embedded systems are typically hidden from the user, i. e., the operator. Embedded computer systems control physical processes according to a given control software. To this end embedded systems interface sensors and actuators which are used to acquire input data and generate output reactions. The operator makes use of the system via a man-machine interface, which is specifically designed for the purpose of the system in question and should be easy to operate [79]. Further, embedded systems typically have neither a sophisticated interface for programming and debugging, nor are they multi-purpose computers. The employed hardware is rather reduced to the minimum necessary for the intended function. This also applies to the processing power and memory capacities of the employed components.

Marwedel mentions in [98] a number of application areas, in which embedded systems are being used. Amongst those areas are:

Consumer electronics: Music and video players, either stationary or portable, are a prominent example of embedded systems. Using digital signal processing, these devices provide an increasingly number of services which are of better quality compared to their analog counterparts. Another major driver of innovation in the area are mobile phones which have a special emphasis on communication and energy-awareness.

Plant-automation systems: One of the first fields for real-time computer control were plant-automation systems. Compared to the former human operators, directly controlling the plants, a computer system is more accurate and dependable, especially in a twenty-four-seven duty cycle.

Automobiles and airplanes: Cars make use of embedded computer systems for the reliable operation of safety relevant functions like ABS, airbag controllers, electronic engine control, etc. Similarly, lots of the control systems of modern airplanes are implemented by embedded systems. This includes flight control systems, anti-collision systems, and pilot information systems.

From the above list it is obvious that embedded systems are used in a variety of systems with real-time constraints. While missed deadlines in consumer electronics devices only reduce the quality of those systems, a similar miss in a plant-automation system, or an automobile or airplane, could endanger the operator as well as other people. Consumer electronics accordingly can be considered embedded soft real-time systems, while the other mentioned systems are clearly embedded hard real-time systems.

According to Kopetz, embedded real-time systems share some key characteristics [79]:

Mass production: Most embedded systems are intended for a mass market. They are produced in large numbers and the production cost of any single unit must be as low as possible.

Static structure: The embedded system is usually used unaltered throughout its lifetime and has a dedicated use. The a priori known static environment und intended function helps in analyzing requirements and designing the software. This avoids unnecessary complexity of the systems, since little flexibility or dynamic algorithms are needed.

Ability to communicate: In case of distributed embedded systems like cars or airplanes, robust and deterministic behavior of the employed communication system is more important than transmission speeds. Therefore, the focus for the employed protocols results in predictable timing and absence of collisions.

The mentioned application areas and characteristics for embedded real-time systems show that they require special care during their implementation. To this end, a comprehensive view onto the overall system is beneficial, especially if it is distributed. We will exemplify how this view can be achieved using model-based approaches in the next section.

2.3 Embedded Real-Time Systems Development

The design of embedded real-time systems is guided by a load of different prerequisites, arising from the system's environment, its intended functionality, and the safety requirements demanded of it. The size and complexity of today's distributed systems, for example in automobiles are expected to reach up to 100 million lines of code in 2010 [116], making development even more difficult. To tackle this complexity, a suitable abstraction as well as a separation of concerns is employed. By using models for the description of the system, details unnecessary for the respective work are hidden from the system developers. The utilization of different types of models enables developers to solve different issues in separate development steps [28, 79].

As explained by Harel [65], the design of complex real-time systems can be eased by separating the functional design, i. e., the behavior of the system from its structure, that is its architecture. The same concept is proposed in articles by Broy [28], and Sangiovanni-Vincentelli and Di Natale [116], who advise the separation of modeling functional design and physical distribution of the software. Figure 2.8 depicts a possible design flow for such a modeling approach. After the functional architecture is designed, the system is partitioned into distributable components. An allocation of these components to their respective location of execution is achieved based on a model of the target platform. The platform is modeled by means of the physical topology of available processing nodes and devices in combination with their respective key figures and the employed communication system.

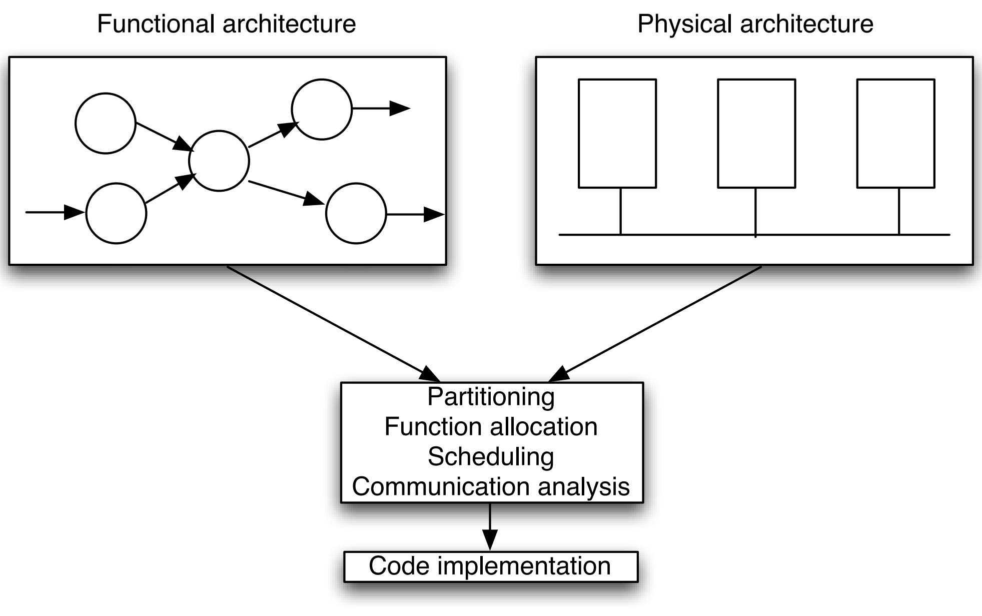


Figure 2.8: Separating functional and physical architecture

Languages based on the so called *synchronous approach* have proven their suitability for the functional design of highly safety critical real-time systems. Some of them were used during the design of flight control systems for several Airbus models, as well as the control systems of nuclear power plants [19]. The synchronous approach eases the separation of functional and architectural designs through its concept of modeling time. According to this approach, the execution of any component is carried out instantly. After its activation by some event, the component is processed in logically zero time. As a result timing model assumes that the components outputs are produced synchronously with the respective inputs — therefore the name *synchronous*, as explained by Beneviste and Berry in [16]. Events can only occur at discrete points in time, called ticks. This view of time as a sequence of clock ticks renders a discretization of the underlying continuous time. The event triggering a component's execution may either arise from the system's environment or it may be a timing event. The synchrony assumption is beneficial for the partitioning of a system into functional components, because of the execution of any component being instantaneous, there is no need for complex timing analysis of the different components during partitioning. Rather, timing analysis is separated from functional and architectural design. After partitioning is finished, the actual WCET of any component is analyzed and matched to the deadline specified for it in order to calculate schedules for the execution on an actual platform.

A more detailed introduction to synchronous languages will be given in Section 4.2. Well-established languages based on the synchronous design are, for example, Signal [17], Esterel [22], Lustre [61], and Statecharts [64].

The mapping of the designed software components requires the availability of a platform model. This model gives an abstract view of the capabilities of the hardware platform from a software developer's point of view. The model should provide data about processing speeds, memory, communication speeds, etc. However, up to now there is no approach which provides functional and platform modeling to the level of detail necessary for generating a distributed system thereof. Instead, modeling languages are well suited to model functionality — like the synchronous languages — or they facilitate modeling of software and hardware architectures. In order to avoid the need for combining several tools which are perhaps not entirely compatible, a language covering both dimensions would be preferable. If such a language is combined with a platform providing standardized interfaces, automated generation of the system would be facilitated.

Historically, the target platform for an embedded systems was often designed from scratch along with its software. For complex distributed embedded real-time systems an effective co-design of hardware and software throughout the entire development process is often implausible due to the sheer size of the system and the number of developers and different tools involved. Instead of a complete redevelopment, a standardized interface between application software and the platform consisting of hardware and basic software like operating system, device drivers, and communication primitives is needed [28, 32, 88, 116]. Such a platform concept leads to an architecture which has to provide robust communication and deterministic behavior by construction [70, 95].

If functional modeling is carried out using a synchronous language, employing a time-triggered platform is ideal, since those platforms fit well with the synchrony assumption, cf. [19]. The discrete ticks of the synchronous software model can be mapped easily to the timing events of the time-triggered platform. Another benefit of a time-triggered platform is its timing predictability. A prominent example for such an architecture is the TTA, which has been used in numerous automotive and avionic projects [80]. We will give an overview of the TTA in the following section.

2.4 Time-Triggered Architecture

The time-triggered architecture is intended to provide a computing infrastructure for the design and implementation of distributed embedded real-time systems. To this end, the TTA defines concepts for distributing real-time applications over a number of computing nodes and providing reliable communication between those nodes. Besides being used in actual systems like the Airbus A380 [95], it has been shown to be a suitable platform for the semantically equivalent execution of systems modeled with synchronous modeling language Lustre [34].

Concept. The TTA is not aimed at modeling the behavior, that is the functionality, of a real-time system. It is rather meant for providing a reliable, fault-tolerant platform for the integration of autonomous subsystems. The TTA prevents negative mutual influence of the subsystems by avoiding conflicts during the access of the shared communication medium. The TTA is based on the following principles [95]:

Consistency: The TTA provides each node of the system with a consistent view onto the system. The current system state apparent to every node is guaranteed to be identical to that of all other nodes. The communication protocol of the TTA ensures this consistency by fault-detection mechanisms as well as optionally redundant transmission using two independent bus systems.

Temporal firewall: Bus access in the TTA is governed by specialized *communication network interfaces* (CNI). The CNI takes temporal control over the communication, thus avoiding collisions at the bus level. The sender pushes its information into the CNI's memory. The CNI forwards this information according to a pre-defined time-triggered schedule to the communication medium. At the receiver side the local CNI stores the arriving data into its private memory and waits for the receiving application to request them. This decoupling of sub-domains prevents applications from sending messages at random points in time and facilitates predictable communication delays.

Composability: Today distributed embedded real-time systems are oftentimes implemented by a large number of developers. The participating developer teams are neither necessarily working at the OEM selling the system, nor do they use the same tools and processes to implement their part of the software. Rather, a group of suppliers develops parts of the system for the OEM [38]. This division of labor is, for example, typical for the automotive domain, where the OEM is mainly concerned with manufacturing the motor and chassis of the car and most of the electronic subsystems are developed by suppliers.

Integrating the different parts into a functioning system is a huge challenge. By defining strict communication interfaces and semantics, the TTA can provide guaranteed integration of nodes into the overall system. The temporal firewall property of the TTA makes sure that the timely execution of the system is not corrupted by a faultily implemented node. Therefore, the stability of a system is not impacted by adding nodes.

Scalability: The combination of the temporal firewall and composability properties lets developers focus on their respective part of the system to implement. The fixed interface in a temporal and syntactic sense, provides the developers with an abstract view of the system. The successful integration of a new subsystem into and existing system is guaranteed by the concepts of the TTA. This eases development of subsystems a lot.

Communication. Communication in a system based on the TTA is provided by the Time-Triggered Protocol. In the TTP access to the communication medium is controlled by a *Time Division Multiple Access* (TDMA) scheme. According to this scheme communication is arranged in recurring rounds. Each round is divided into communication slots, each of which may be assigned exclusively to one node of the system. Hence, in the respective slot only the assigned node is allowed to send data, while all other nodes have to remain quiet and receive data.

The exact timing is crucial for the correct operation of a TDMA-based communication system. Letting the application processor of each node handle the resulting workload is not advisable. First, it would decrease the processing capacity available to applications by a large amount, not only when data have to be sent, but also when data may be received from other nodes. Second, the processor would have to preempt application tasks in order to execute the communication protocol stack for sending or receiving data. Depending on the message size, the time for which the application task is preempted may vary and thus its deadline could be missed. To avoid the mentioned interference of application tasks and TTP tasks, the TTP is implemented in a dedicated communication controller. This controller is solely responsible for sending and receiving data according to the TDMA scheme. The particular scheme for a system is defined prior to the system's execution and loaded into the communication controller. Messages are exchanged between applications and the controller using a dual-port memory. This memory is accessed by the application processor as well as the communication controller.

The TTP also features clock synchronization for the system. The availability of a global time is necessary to enable operating the TDMA scheme. As the sending slots of all nodes in the system are predefined and known by every node, the difference between the time a message is received at a node and the time it was expected to be received according to the TDMA scheme, can employed to calculate the drift of the local clock. The result is used to synchronize the local offset in time to the global clock.

In summary, the TTA provides the desired abstraction to ease the development of complex distributed real-time systems. The approach of time-triggered communication has been implemented by other bus protocols as well, which are also already used in automotive systems. Rushby gives a good overview of the advantages of time-triggered bus protocols suitable for safety-critical systems in [115]. According to Kopetz, however, TTP provides better reliability at reduced cost compared to FlexRay [81]. Maier in addition states TTP's advantages compared to tt-CAN in [95].

2.5 Chapter Summary

In this chapter we have given an introduction to the basic requirements of real-time systems and the challenges during their development. A categorization into soft and hard real-time systems has been given, as well as a comparison of event-triggered and time-triggered communication. We have argued why time-triggered communication and static scheduling are more suited for our target domain, namely automotive systems.

Considering suitable abstraction concepts for the ease of automotive systems' development, we have presented synchronous languages as a well-established modeling approach. Further, we have introduced the TTA as a concept for providing a standardized execution platform, enabling the distributed development of a system. The TTA ensures composability of the originating subsystems, even without comprehensive knowledge of the overall system available to all developers. The TTA is also a well suited target platform for the deployment of synchronous models.

A common shortcoming of approaches like Lustre and TTA is their focus on isolated aspects of real-time systems. While Lustre is well suited to model a system's functionality it does not consider architecture or distribution aspects. It provides an abstract way of designing a system's behavior, yet it gives no guidance on how to deploy the model onto a distributed platform. The TTA in contrast, is solely meant for providing a basis for successful integration of already written application code. It does not feature any concept for modeling system behavior. A development concept providing comprehensive solutions to both aspects — behavior and architecture of a system — is proposed in this thesis.

Before giving an overview of our approach in Chapter 4, we want to give a more detailed description of the specific needs of automotive systems, the current state, and the future of automotive platforms. The next chapter presents these needs and outlines recent architectural changes for automotive systems, which our approach takes care of.

Chapter 3

Specifics of the Target Domain

We have given a general introduction to embedded real-time systems in the previous chapter. Automotive systems clearly fall in that domain, but compared to airplanes or plant-automation systems cars have evolved differently over the years. The automotive domain faces more stringent constraints regarding cost and weight, and is affected by a different division of labor between OEMs and suppliers.

An overview of the specific challenges of automotive systems is given in Section 3.1. Software engineering is employed — at least in parts of the automotive domain — to tackle these challenges. We will summarize some of the efforts and the remaining problems in Section 3.2. The AUTOSAR standard, which will be explained in Section 3.3, is one of these efforts and is currently being adopted for more and more ECUs. Finally, in Section 3.4, we want to give an outlook on future trends and challenges in automotive systems.

3.1 Characteristics of Automotive Systems

The importance of software for automobiles is undeniable. According to Hardung et al. [63], automotive electronics account for 90 percent of innovations in automobiles. Of those innovations 80 percent are achieved by software. Hardung et al. also mention a study, according to which the cost of software is expected to raise from 4 percent of the overall vehicle cost in 2000 to 13 percent in 2010.

Being of such importance, the quality of automotive systems is a major concern for any OEM. Especially today, as they are no longer limited to ignition or comfort electronics, but include safety-relevant systems for active safety and driving assistance [89]. Moreover, X-by-wire applications have long been, and are still, expected to replace the mechanical control solutions for steering and braking.

Yet, automotive software is far from being error-free. According to a study of the IBM Institute for Business Value, automotive OEMs pay over 500US$ of warranty costs per vehicle [7]. The study suggests that over 30 percent of that cost are caused by electronics and software defects. This lack of quality arises from the continuing effort to produce complex, safety-critical software under tight cost constraints.

An introduction to the specifics of automotive systems is given in the following.

3.1.1 Complexity

The complexity of automotive systems is caused by two distribution aspects: first, the computing platform is composed of a huge number of subsystems or computing nodes. Second, the application software executed on an automotive platform is distributed over its hardware platform. This distribution requires reliable communication and compatible interfaces of the involved tasks for providing the desired functionality. At the same time, hardware and software distribution must not affect the timing requirements of the application in a negative way, causing the application to miss its deadlines. The reasons for hardware and software distribution are as follows:

Hardware distribution. Traditionally, automobiles have been mainly mechanical systems. Software has made its way into cars about 40 years ago. First, it was used in ignition systems, later on its use expanded to airbag controllers, anti-lock braking systems, etc. As described by Venkatesh et al. in [125], these systems operated rather isolated from each other and used their own respective sensors and actuators. With the increasing number of subsystems and the growing number of sensors and actuators, a lot of cabling was required. This cabling accumulated to a lot of weight — up to 50 kilograms [89] — and consumed a significant amount of space. The solution to handle these problems came in form of bus systems, connecting the subsystems to their devices. As indicated in Figure 3.1, the amount of cabling can in many cases be reduced significantly by using a bus instead of a star topology. Naturally, the idea grew that the employed bus connections could also be used to exchange messages between the subsystems, thereby influencing each other. Today, up to five separate bus systems — which are even using different protocols — connect up to 80 ECUs and the necessary sensors and actuators in a luxury car [26].

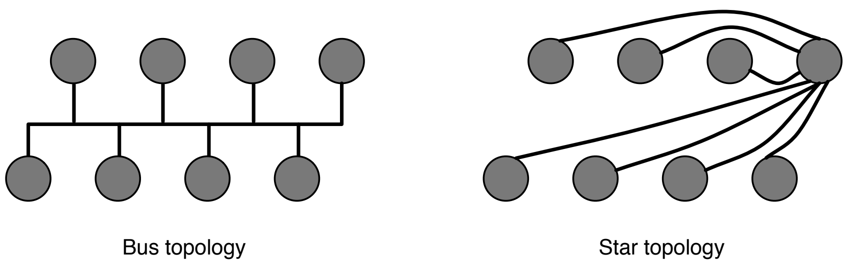


Figure 3.1: Comparison of bus and star topologies

The historical growth of automotive systems is however just one reason for their highly distributed nature. The relationship between automotive OEMs and their suppliers is also responsible for the current automotive architecture. Automotive OEMs see their typical field of work in the mechanical domain of a car and not so much in electronics or software. Therefore, the development of ECUs is distributed over a number of suppliers capable to deliver the desired part. The supplier develops hard- and software fulfilling the requirements specified by the OEM and delivers the result as a black box. The integration of all the different black boxes is then the duty of the OEM [30, 63, 108]. Only few of the ECUs contained in a car are developed by the OEM itself — usually those being intended to differentiate the own products from those of other OEMs. For all other electronics the trend of outsourcing development continues [38].

Further, parallelization is a striking argument for the employment of several ECUs — and thus processors — compared to a centralized in-car system. By using separate processors for different time-critical applications, their timely operation is easier to ensure. Today, this advantage is partly neglected by the distribution of applications over several processors. The implicated bus communication and its delay are hard to predict due to congested bus systems. The automotive industry handles this problem by using bus systems with more and more bandwidth. In addition, several bus systems are employed, which are interconnected by gateways [103]. This adds further complexity to the distributed automotive topology.

Finally, the different nodes can be used as fault-containment units. If the application execution on a node produces errors or the hardware of the node fails, a distributed execution allows to isolate the resulting error state from other nodes of the system. Therefore, the failure of one node does not necessarily compromise the operation of the entire system [95, 107]. However, for distributed applications this may not be applicable.

Software distribution. As described before, the availability of a communication system led developers to design applications which exchange messages during their operation. This data exchange enables the use of hardware for several different applications. A wheel speed sensor, for example, can be used for an anti-lock braking system as well as for electronic stability control and a tire-pressure monitoring system. These applications are not necessarily implemented in the same computing node. That is why the exchange of data is necessary. Many functions provided by an actual car are distributed over a large set of nodes. A common example for a highly distributed application is the central locking system, which is implemented using up to 19 different subsystems in current vehicles [125]. This increased coupling between distributed components results in very complex and manually hard to implement application code [30, 52]. This becomes even more evident regarding the huge amount of code used in current automotive system, amassing up to 65 megabytes of binary code [127].

Given the described physical distribution of the hardware platform as well as its high interconnection by distributed applications, the development of high quality automotive systems is a huge challenge. According to Venkatesh et al. [125], the steady increase in the number of hardware modules employed was accompanied by an exponential increase in interaction of these modules.

3.1.2 Safety Requirements

With the high amount of interaction between different features, ensuring the safe operation of the overall system is very demanding. Compared to avionic systems, which were first used in military planes and faced a high demand of safety, automotive software was in the early days limited to rather noncritical applications like ignition control systems. As a result, neither safety considerations were applied nor are systematic development processes required by law [125].

Even today, process standards which are intended to enhance safety of an automotive system are optional. The use of such a process still does not necessarily result in a safe system. Instead, the explicit use of methods for safety analysis is required [118, 127]. Of course, the system's implementation has to avoid errors where possible. MDD has shown to effectively reduce the number of errors and, consequently, results in a higher quality and often also safer system [90].

Taking into account the usual system lifetime of 10 to 15 years in the automotive sector, avoiding any implementation faults is even more vital as errors are more likely to show up during such a long operating time [28].

3.1.3 Cost Constraints

Compared to the avionic domain, the automotive domain faces a much higher cost pressure on a per-unit scale. While 100 to 1,000 units of a specific aircraft might be produced, a car manufacturer produces 50,000 to 1 million units of a model [28]. In a high volume domain like the automotive one, the addition of a small amount of memory or the use of a faster processor in one of the ECUs may add up to a large amount of money [122]. Therefore, automotive systems face more narrow resource constraints than avionic solutions. Still, an enormous increase of hardware and software could be observed over the past years. These systems are expected to account for up to 30 percent of a car's overall cost in the near future [26]. However, the effort to drive down this cost by cutting processing speeds and memories is not without problems. As a result, code has to be optimized for the respective ECU and can hardly be expanded with new functionality or reused for another platform. Moreover, in case of failures it is harder to identify and fix the bugs [30].

Another major cost factor can be found in form of development cost. Regarding the high number of units produced, development cost per unit are much lower in the automotive domain than in the avionic domain. But with the increasing amount of functionality realized by software, development costs keep growing as well [30]. Therefore, automotive OEMs try to decrease development expenses where possible in order to manufacture products at competitive prices. MDD is seen as one of the most promising approaches for reducing development cost and time to market while at the same time increasing product quality [90]. On the one hand, the possibility to quickly implement new functionality is a key factor in competition with other OEMs, especially in the luxury class [52]. On the other hand, the improved quality reduces warranty and goodwill costs caused by erroneous systems.

Cost reduction can also be achieved by shortening development cycles via reusing existent code. However, the predominant approach of the OEMs, which considers all ECUs as separate units, hinders the reuse of software. ECUs are ordered from the suppliers as black boxes, consisting of soft- and hardware. The particular code is implemented specifically for the respective hardware. Reusing that code for a new project, possibly demanding different hardware and being developed in cooperation with another supplier, is impossible [63]. This is why the reuse of code calls for a common platform interface. Implementing such an interface in hardware would result in an inflexibel solution. A standardized software interface however can be implemented on a lot of different hardware platforms. One automotive platform concept that features the needed abstraction is presented in the Section 3.3.

3.2 Current State of Automotive Software Engineering

Naturally, the development of automotive software has to adhere to the afore mentioned constraints. The historic relationship between OEM and supplier, where the supplier builds a mechanical component of the car and the OEM uses these components to assemble the final product, has been adapted to automotive software development. Today, with the software — and often also the related hardware — being developed by suppliers, the OEM is responsible for integrating the supplied subsystem into an automotive network. But, as design, architecture, and interaction are only specified insufficiently due to lacking experience with software development, integration is error-prone and requires lots of error correction [27]. Lately, there has, for some part, been a shift in this classical division of labor. Since functionality implemented by software has become a major part of automotive systems, OEMs have begun to develop an increasing part of their systems in-house. Thereby they can ensure the exclusivity of a function for their automobiles, which allows to distinguish themselves from the competition [52].

No matter whether parts of the system are developed by the OEM or an supplier, regarding the huge dimension of nowadays' automotive systems, development time and software quality may significantly improve using clear specifications. For this reason, the automotive industry has begun to use models and model-based development, even if only at particular areas in the development process. The employed modeling approaches are however still influenced by the various core disciplines involved in automobiles, such as electrical and mechanical engineering. Each of the disciplines, as well as each supplier, is free to use its own process and tools for development. This results in a large number of different tools from different vendors, lacking a common modeling concept. There is still no tool-based solution covering the entire development process and yielding a comprehensive model of the overall system.

As a consequence the gaps between phases and tools have to be covered manually by porting data from one tool to the next which is error-prone and time consuming. Attempts have been made to form a tool-chain from existing tools. But the results are not yet ready for production [52, 108]. With modeling languages like the UML employed, which have no precisely defined semantics, the use of models is at the moment mainly restricted to structural modeling and documentation purposes [21, 45]. The derivation of consistency checks or target code from such a model is hardly possible or leads to ambiguous results. The lack of automated code generation is an even bigger problem, regarding the percentage of problems caused by coding errors, falling between 40 and 60 [128].

Another shortcoming can be found in the lack of a capable standardized platform. Today, 90 percent of software for automotive systems is rewritten, although only 10 percent of the functionality is actually changed from one car generation to the next [27]. This is mainly due to a lack of compatible execution platforms employed in different car generations. Even though the automotive industry has created specifications like the OSEK[1] standard, defining the interfaces an automotive operatings system and communication stacks should provide, a comprehensive standard that takes distribution of the platform into account, is missing. OSEK also does not define any interfaces for accessing hardware devices. The latest attempt to close this gap has been made in form of the AUTOSAR standard, which provides a standardized execution platform, as well as a concept for architecture

[1]http://portal.osek-vdx.org

modeling. Currently, the industry is moving towards a migration of its systems for use with AUTOSAR. The AUTOSAR concept however has also some shortcomings from a software engineering point of view, as we will explain in the following section.

3.3 AUTOSAR

The design of the Automotive Open System Architecture was initiated to define an open industry standard for automotive architectures. It was started by a number of automotive manufacturers and their suppliers, among which are BMW, Toyota, General Motors, Ford, Volkswagen, Daimler, and Bosch [46]. AUTOSAR is aimed at decoupling application and infrastructure software. To this end it comprises the AUTOSAR metamodel which describes a schema for models of automotive systems. Semantics of the metamodel are only partially defined, giving users the possibility to tailor the use of AUTOSAR to their respective needs [75]. AUTOSAR also defines a platform interface which is provided by a common software infrastructure [66]. This infrastructure includes operating system, communication primitives, and drivers for hardware devices. It is also referred to as *basic software* in the AUTOSAR documents. Figure 3.2 shows the generic layout of an AUTOSAR compliant architecture.

AUTOSAR provides the application software with the so-called *Virtual Functional Bus* (VFB), which serves as a mediator for communication between application software (AUTOSAR software) and infrastructure (basic software), as well as between different application software components. The VFB is a model of the platform's communication system and is only present in the AUTOSAR model. For an actual node of the system, it is implemented by the AUTOSAR *Runtime Environment* (RTE), shown in Figure 3.2. The RTE carries out the mapping of the VFB's transparent communication onto a distributed hardware platform.

For automotive systems it is common to use sensors and actuators from different vendors for the same model of a car. This results in a higher competition between the suppliers which lowers the price. Further, the availability of a sufficient number of hardware devices can be ensured. From a software point of view this causes a problem. Different devices often feature varying characteristics or interfaces requiring changes in the respective application code. In order to deal with this problem, AUTOSAR software consists not only of application software components, but also of sensor and actuator software components, as depicted in Figure 3.2. These sensor/actuator components provide a common mode of operation to application software components no matter which version of a device is used. To integrate different devices into the platform it is sufficient to replace the sensor or actuator software component. The application software component may be used unaltered, which improves reusability of the application code.

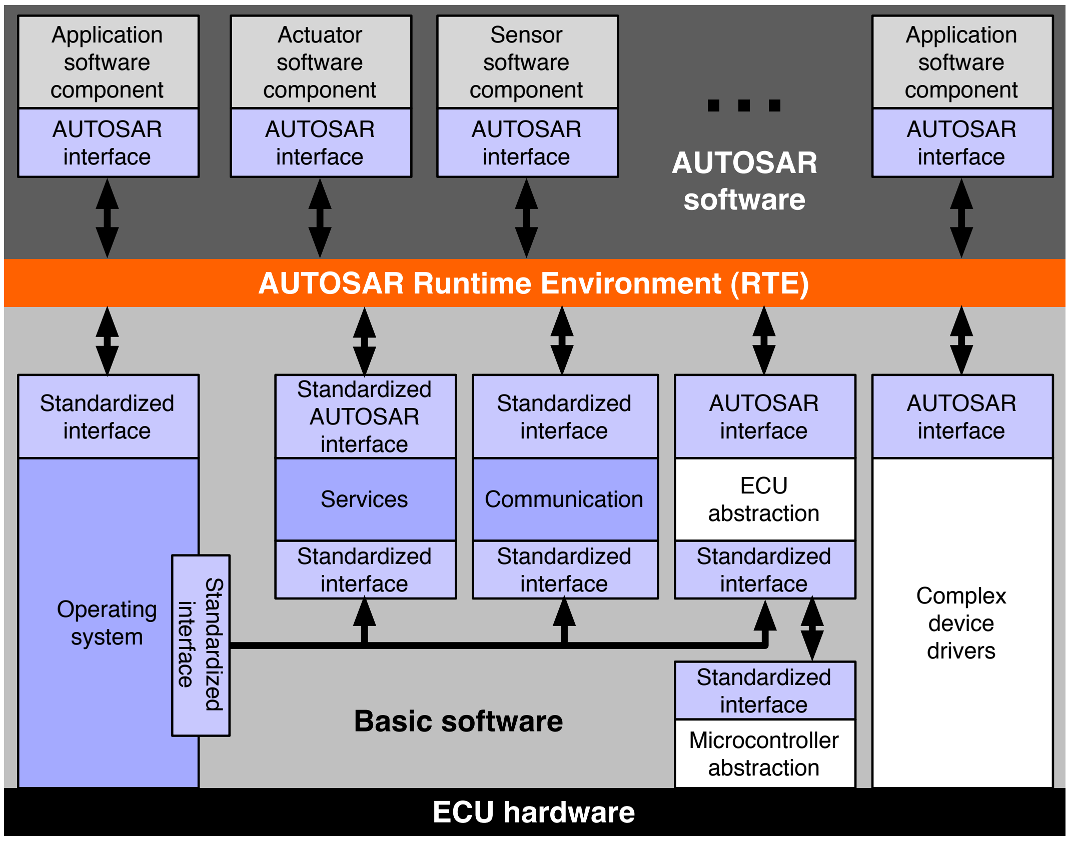


Figure 3.2: Basic structure of an AUTOSAR system (cf. [8])

Main goals of the AUTOSAR standard are the improved reusability of software by using a common platform and standardized interfaces, and provision of architecture descriptions for the distributed development of software components and their subsequent integration.

3.3.1 AUTOSAR Modeling

According to the metamodel, an AUTOSAR design consists of models for application *software components* (SWC), ECU resources, and system constraints defining a mapping between the two. For all these models templates for suitable exchange formats — based on XML — are available. Thus, the OEM is enabled to pass a defined architecture, or parts of it, to suppliers and distributed implementation is facilitated. AUTOSAR does not specify a universal software development process and this is the reason it lacks a strict order of activities [126]. The definition of such a process is up to the respective OEM.

An *AUTOSAR software model* consists of SWCs which can be modeled either as *atomic* or *composite components*. While composite components can be aggregated from other SWCs, atomic components do not contain further SWCs. Rather, an atomic SWC is made up of a set of *runnable entities*. A runnable entity encapsulates some functionality which is triggered by some *event*. As soon as the event occurs, the runnable entity is started and executed to completion. The SWC may define shared memory for the runnables it contains. To specify input and output connections to the SWC's environment, the component exhibits *ports*. These ports are connected to ports of other components by means of *connectors*. Communication between SWCs is only allowed using these ports and connectors.

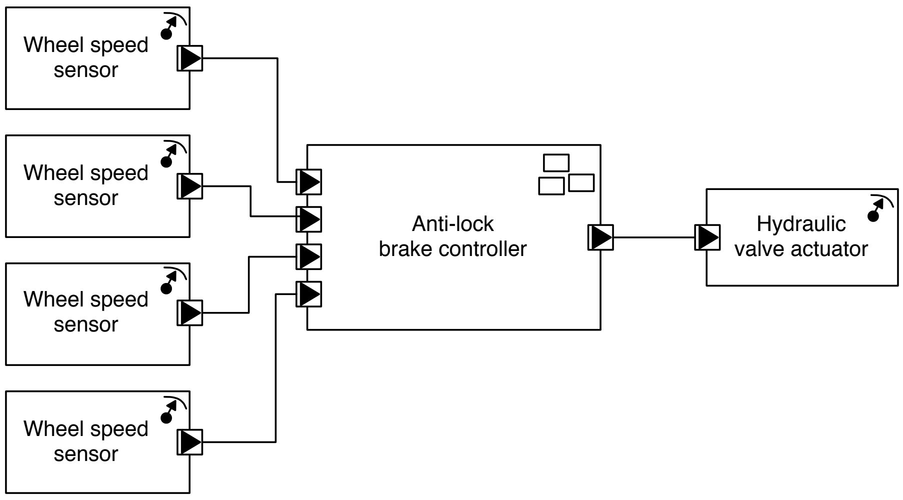


Figure 3.3: AUTOSAR model for the anti-lock braking system

Figure 3.3 shows a possible AUTOSAR software model for the anti-lock braking system from Figure 2.2. In Figure 3.3 six SWCs can be seen. The SWC named `Anti-lock brake controller` implements the control-loop of the ABS and is a composite component, indicated by the pictogram in the upper right corner. The other five SWCs are sensor and actuator SWCs, respectively. Hence, they share another pictogram. All software components in the model feature ports at their boundary lines. Ports pointing inwards are receiver ports, while ports pointing outwards are sender ports. The ports are concatenated using connectors. The example gives only a brief impression of AUTOSAR's modeling elements. A complete overview is given in the respective documents of the standard.

An *AUTOSAR hardware model* consists of sensors and actuators and ECU instances connected by physical channels. The channels correspond to the bus topology of the actual hardware. This model is employed to define a mapping of the modeled application components to ECUs.

In order to provide a decoupling of application SWCs and an actual hardware platform, the VFB serves as an abstraction layer. All external communication of a SWC is mapped to the VFB, which abstracts the application layer from the implementation details of the basic software as well as hardware aspects. The VFB forms this way the interface to the execution platform, from the applications point of view.

3.3.2 AUTOSAR Platform

When the design of the system is finished, code generators may be used to translate the VFB model into the so-called Runtime Environment. A specific RTE instance is created for every ECU of the system and serves as a middleware layer at runtime. The RTE is not a classical middleware, providing dynamic look-up of services or registration or additional applications at runtime. It can rather be thought of as a model-based middleware [108], since the provided mapping of the application interface to the common software infrastructure is hardcoded during code generation. For an automobile this approach is valid, since in actual automotive systems the after-market installation or upgrade of software is not an issue nowadays. The advantage of the RTE's fixed structure is its small footprint regarding memory consumption and processing overhead.

In order to facilitate the RTE generation, AUTOSAR also defines a set of standardized interfaces which have to be provided by the basic software. This is particularly crucial for the API of the employed operating system and the communication system. The necessary operating system interface is based on the OSEK specification [9]. The goal of the communication system API is to define a common interface for exchanging data with other ECUs, no matter which bus protocol is used.

3.3.3 AUTOSAR Open Issues

AUTOSAR does not solve all of the previously mentioned problems of automotive software engineering. While it does define standards for modeling systems and basic software interfaces, there is no reference implementation available. Rather, the specific implementation of the standard is left to the user. Unfortunately, the standard is not described unambiguously in the AUTOSAR documents. As a result, different OEMs and tool vendors have developed their own respective implementations. Because AUTOSAR does not define a comprehensive data model, consistency of models is not guaranteed when designed by different parties involved in the development of a system [126].

Furthermore, designing an AUTOSAR system is restricted to architecture modeling. The exact behavior of SWCs — or, more precisely, runnable entities — is not contained in the model. That is why it is not possible to model-check, generate code for, or simulate an AUTOSAR model in a comprehensive way, although this would ease debugging and result in higher quality systems.

Another issue is that timing semantics of an AUTOSAR model are not clearly defined. AUTOSAR allows the definition of event-triggered and time-triggered scheduling models, as well as combinations of both. In addition, the buses which may be used for the platform may also be event-triggered or time-triggered. If event-triggered tactics are used, the timing analysis based on an AUTOSAR model is at least imprecise, if not impossible.

3.4 Future Trends in Automotive Systems

With the number of functions implemented by automotive systems ever increasing, the complexity of their hardware and software architectures has reached a dimension which is difficult to handle.

Considering the hardware architecture, the sheer space consumed and weight accumulated by adding more and more ECUs to automotive systems has become a major problem. Besides, the large amount of wiring also compromises reliability, as the large numbers of connectors are more likely to fail [89, 119]. On the other hand, more powerful processors become cheaper over time, making it possible to execute a larger number of applications without increased hardware cost. This trend already allowed the avionics domain to rely on a smaller number of computing nodes, each of which has higher processing power. Even the overall computational power of the system could be improved. Each of the nodes in such a system integrates functions from a number of the legacy nodes used in former architectures. The platform concept is called Integrated Modular Avionics (IMA) [24, 109]. A similar trend of using a smaller number of more capable ECUs is imminent in the automotive industry [107].

The use of less computing nodes could also alleviate the problem of current automotive communication systems. Because of the large number of communication partners and their different criticalities, automotive platforms usually comprise several communication networks. This use of a dedicated network per domain ensures higher capacity and fault-containment compared to a single bus system. However, since the different networks contain applications which exchange data, they are interconnected by gateways. These gateways are rather powerful devices, since they have to handle large amounts of data. With faster, predictable bus protocols available, like FlexRay or TTA, it is possible to fulfill the capacity and fault-containment needs within a smaller number, if not a single bus [103].

As another result of the extensive use of electronics in automotive systems, energy consumption has risen dramatically. Well-equipped automobiles draw 2kW and more peak power [89]. In case of momentary peaks, the battery serves as a buffer for providing a stable voltage supply. This, however, significantly reduces its lifetime. Moreover, the electric generator requires a reasonable amount of mechanical power from the motor, which increases fuel consumption. The issue becomes even more critical considering the current shift of automobiles towards electric driving, where a lower energy consumption of the overall systems means higher operating range.

Reducing the amount of energy consumed in an automotive system could be achieved by deactivating unneeded ECUs, as well as sensors and actuators. But such a fundamental change in system behavior could compromise safety through unintended side effects, if not planned carefully. Designing such an energy-aware system would require detailed knowledge about all applications executed and the hardware employed. Nowadays, such a comprehensive system model is not available and complex applications affecting the overall system are almost impossible to realize. As a consequence, future modeling methods have to provide a comprehensive view of the system, including architecture, behavior, timing, and hardware characteristics [27].

3.5 Chapter Summary

In this chapter the specific challenges of automotive systems and their implementation have been introduced. As explained, the number of computing nodes included in an automotive system has grown continuously over the past 40 years. Implementing such a system is a huge challenge regarding the complexity of current automotive software. Even more so when taking into account the high quality demands arising from the safety-criticality of the implemented functions.

In order to tackle the complexity, model-driven development is seen as a possible solution. By using the abstractions provided by a model, developers are able to concentrate on a smaller number of concerns at a time, which results in shorter development time and higher system quality. Unfortunately, MDD is still only used for parts of the system. Moreover, a multitude of different modeling tools and languages is used throughout the development process, which hampers the integration of the different subsystems.

With the advent of the AUTOSAR methods, a standardized modeling concept for the automotive industry was created. But even during AUTOSAR development incompatibilities may arise, due to differing customizations of the respective AUTOSAR models. Furthermore, AUTOSAR is not intended to model system behavior, i. e., functionality, but is limited to architecture modeling. Lacking a possibility to model and analyze the exact timing behavior of a system may result in systems which do not meet their timing requirements. And the AUTOSAR approach still requires a lot of manual work necessary for implementing an overall system, possibly leading to erroneous results.

Future automotive systems are subject to a reduction of processing nodes and an even higher number of software functions. Their implementation is therefore even more critical since a negative mutual influence of different coexisting functions on a computing node must be prevented. This calls for new modeling methods, which provide a more comprehensive view onto the system under design and extended tool support during implementation. Consequently, the number of human faults during system implementation may be reduced.

Chapter 4

The COLA Approach

As described in Chapters 2 and 3, automotive systems have huge quality demands regarding their real-time and safety requirements. At the same time the development of those systems has to account for the huge cost pressure in the automotive domain, without compromising these requirements. Thus, methods for the efficient development of high quality automotive systems are sought-after.

In this chapter we will outline how the modeling language COLA, and the development approach based on it, helps to achieve these goals. Section 4.1 gives an overview of the proposed process and states its advantages compared to existing approaches. Afterwards, in Section 4.2 the basic concepts of synchronous dataflow languages will be introduced to provide a basic understanding for the mode of operation of the COLA language. Subsequently, in Section 4.3 we will introduce those concepts and language constructs of COLA, which are necessary for the understanding of this thesis.

Besides the deployment plug-ins described in this thesis, several other tools have been designed and implemented for the COLA language throughout the project. Section 4.4 presents these tools and their use for the design of COLA models. Since the development of automotive systems is not an entirely new domain, a number of tools and approaches has been proposed over the past years. A survey of existing tools and their differences in comparison to the COLA approach will be given in Section 4.5. Since COLA is at the core of our concept, we will limit the description of related work to alternative modeling approaches. Additional references related to our deployment concept are given in the respective sections.

4.1 Our Vision for MDD of Automotive Software

Model-driven development is seen as the most promising way to get to grips with the complexity of distributed hard real-time systems like automotive networks. The abstraction provided by models lowers the complexity of the system apparent to the developers and thus effectively decreases the number of design faults. However, there is not a single tool providing an integrated approach for both, vertical and horizontal dimensions of automotive development. By vertical we refer to the consecutive refinement of a model during the development process, using different abstractions — with its levels of detail — throughout the respective development phases. The horizontal dimension, in contrast, refers to the distribution of automotive applications over a large number of processing nodes. Since development in both dimensions is split among a lot of project groups, oftentimes working for different sub-contractors, a comprehensive, integrated development process is still rather an exception to the rule. Up to now there is no comprehensive tool available yet for these reasons.

The tools commercially available are instead limited to the description of selected aspects of an automotive system, ranging from requirements to its architecture, the specification of its functionality, or the employed hardware components. Throughout the automotive development process however, the use of a multitude of different tools is inevitable today [63, 92]. The result is a manual conversion and integration of different data formats between the employed tools. Of course this practice is error-prone and hampers reuse and refinement of models. As a workaround it has become a common technique to implement adapters automating the necessary data conversion steps. But this workaround yields large implementation and maintenance overhead, and cannot guarantee a lossless transformation, if the employed modeling languages are of uneven capabilities. Further, a comprehensive view onto the entire system enables the use of global optimizations, rather than applying improvements just locally. To tackle this problem, the COLA approach has been developed in cooperation with BMW Research and Technology [55]. This new approach was designed to cover the entire automotive software development process using a single modeling language. In addition it facilitates the automated transformation of the model into an executable distributed hard real-time system. To this end the approach we envision is a combination of modeling and platform concepts.

At the core of the COLA approach is the Component Language. This modeling language covers all software relevant information from requirements, over software architecture, to functional design, and technical platform details. At the same time it takes the distributed nature of the target platform into account, enabling the automated code generation for, and configuration of, the target system. The implemented COLA editor features a set of plug-ins which enable the automated processing of the model to raise its quality and finally derive an executable system. We will describe our requirements for a suitable execution platform in Section 5.2. Figure 4.1 gives an overview of the different actions throughout the COLA-based development process, with a special focus on tool supported actions. We will describe each of these actions in the following.

System modeling: The process starts with the definition of a system model, which is indicated in the upper left of Figure 4.1. COLA facilitates the modeling along different layers of abstraction, as we will detail in Section 4.3. This results in a comprehensive system model containing all information about soft- and hardware which are necessary for deriving an implementation of the software system.

Model analysis: The formal semantics of COLA enable the application of model checkers for an analysis of the model's correctness. This leads to a reduction of faults in the functional model, with respect to the defined requirements as well as the detection of possible modeling faults like type incompatibility, non-determinism, and the likes.

In order to estimate a feasible complexity of the model regarding the employed target hardware, it is beneficial to analyze the performance needs already at model level, that is before any deployment takes place. To this end, the simulation of COLA models using SystemC has been developed, as described by Wang et al. in [129]. If the simulation indicates that the model is overly complex, either the model or the hardware platform have to be modified accordingly.

System partitioning: The checked model serves as input for system partitioning which divides the overall software model into distributable entities. The partitioning can either be carried out manually by the developer or calculated automatically according to some other heuristics like a maximum size constraint.

Code generation: Starting with the partitioned model, the code generator produces a source code file for each distributable entity, which may then be compiled to derive an executable program. A middleware assures the necessary communication at runtime.

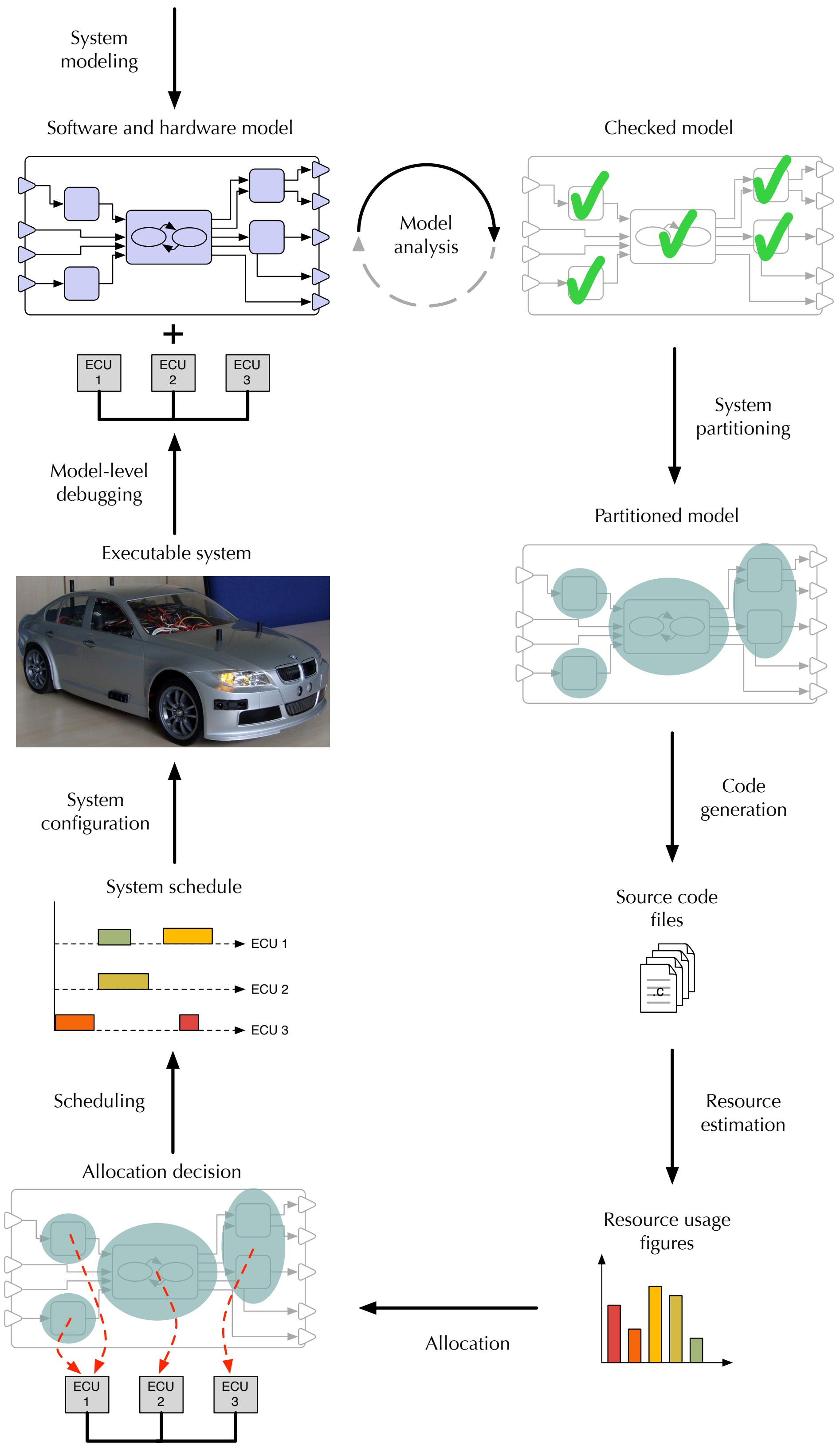


Figure 4.1: COLA development workflow

Resource estimation: The generated source code is compiled and used by the SciSim tool, which has been described by Wang et al. in [130], to derive cycle accurate execution times for each microprocessor in question. The obtained resource figures serve as an input for the subsequent allocation and scheduling steps.

Allocation: When the resource consumption of all tasks is known, a valid allocation scheme can be calculated. The according algorithm is based on the use of non-functional requirements which may include, besides computing and memory resources, further parameters like deadlines, energy consumption, redundancy, etc. The algorithm has been presented first in [85].

Scheduling: When a valid allocation has been identified, a possible schedule for this respective allocation is calculated. The schedule respects all computing times, communication delays, and possible jitter. If it is impossible to find a valid schedule for the current allocation, the allocation is marked invalid and a new allocation must be computed. This round-trip continues until a valid pair of allocation and system schedule has been found. We have introduced our scheduling approach in [84].

System configuration: With allocation scheme and schedule available, the execution platform may be configured. To this end configuration files for the employed middleware as well as the task scheduler are generated. These files assure the correct integration of the modeled system at target level. The result is an executable system which matches the behavior modeled in COLA.

Model-level debugging: The final system might still contain bugs which could not be identified during model checking. One reason may be checks which cannot be carried out due to their computational complexity. Another source for faults is environmental feedback which does not meet the assumptions made in the model. To facilitate the debugging of such design faults the COLA approach optionally features the concept of model-level debugging. This concept enables the mapping of runtime data back to the COLA model, making it easier to identify the source for errors in the model and fix the specific mistake. We introduced this concept in [54].

It is important to point out that all described steps from code generation onwards are performed in an unattended manner by the COLA tool-chain. In addition, model analysis is conducted automatically and system partitioning may optionally be calculated without the input of a system designer. The generated application code together with the system configuration data yields an executable system, without the need for further manual integration effort. This high degree of automation eliminates a large amount of possible sources for errors during distributed systems development. Hence, the described process covers the previously mentioned horizontal dimension in the development of automotive applications.

The vertical dimension is taken care of by the integrated nature of the COLA language. Comprising all modeling steps from requirements definition, architecture description, functional specification, to technical realization, COLA avoids the error-prone porting of model data from one tool to the next. The resulting systems are of higher quality and contain less faults. To retain this quality down to executable code, an automatic deployment concept like the one introduced in this thesis is a necessary prerequisite, as Sangiovanni-Vincentelli and Di Natale postulated in their work about system design for automotive applications [116].

4.2 Introduction to Synchronous Dataflow Languages

At the core of the COLA approach is the Component Language which is a synchronous dataflow language. Before we delve into the details of COLA in the next section, we want to give an introduction to the concepts of synchronous dataflow languages and the reason why they are suitable to model automotive systems.

In order to be useful for the design of automotive — or more general, reactive systems — any modeling language used provides an abstraction from specifics of the actual execution platform, rather relying on higher-level logics. Further, reactive systems share several key features, as described for example by Halbwachs et al. in his introduction to the synchronous dataflow language *LUSTRE* in [61]. These features have to be taken into account during design of such a system. Let us recapitulate these characteristics, which have already been mentioned in Chapter 2, due to their importance:

Parallelism: Reactive systems are often parallel systems in order to achieve the necessary reaction times for a timely interaction with their physical environment. Besides that the employment of a distributed solution may provide better fault tolerance.

Reliability: Reactive systems are used in safety-critical domains, like the automotive domain we are dealing with in this thesis. The correctness of their design is therefore crucial for the safety of the system. The final system's quality may be improved, if the modeling language supports formal verification methods and automated code transformation.

Time constraints: The proper operation of a reactive system is not only based on its result values, but also on the moment in time at which these results are produced. A delay during evaluation of the given tasks may reduce the use of the result or may even render it useless.

In the following we will argue why synchronous dataflow languages are suited to address those requirements.

4.2.1 Dataflow Languages

As summarized by Ackermann in [2], dataflow languages have originally been invented to ease the complex programming of multiprocessors, vector machines, and array processor computer systems. Earlier programming languages for those architectures revealed the hardware properties of the computer to the programmer, aiming at an efficient use of its capabilities. However, such a program is specific to a certain machine and is possibly hard to understand, program, and modify without in-depth knowledge of the target processor architecture. Dataflow languages where created as a higher level programming concept independent of hardware specifics and can be mapped to parallel hardware architectures automatically by the respective compiler.

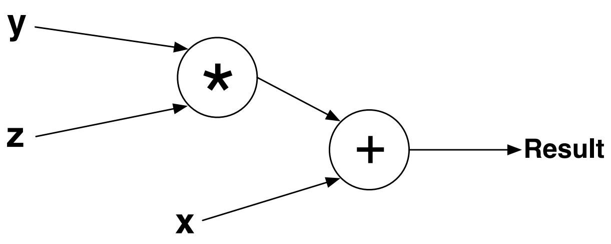


Figure 4.2: A simple dataflow model

The graphical representation of a dataflow program is usually a directed graph, where edges — or vertices — depict the flow of data and nodes represent operations on these data. The nodes are — depending on the respective language — referred to as *operators*, *functions*, or *processes*. Edges transfer input and output values between operators. The causal order of operators is implied by the edges, that is, an operator may only be applied to the input data when all values are available. Applying the operation makes the result(s) available at the outgoing edge(s). Operators may not use any other data than those provided by incoming edges. Figure 4.2 shows an example for a dataflow network which calculates the result of $\mathtt{x} + \mathtt{y} * \mathtt{z}$.

The example shows what Hils denotes in his survey of dataflow visual languages [73] as *pure dataflow*. Such a language does not include any control flow elements like loops or selection/branching statements. According to Hils, designers of dataflow languages often find it necessary — depending on the target domain —

to add constructs for control flow to the language, like SWITCH-statements. Control flow constructs enable the developer to specify distinct execution paths in a convenient way. The same is true for COLA, which features *automata* to express control flow, as we will explain in Section 4.3. Each state of the automaton incorporates a different execution path which may be chosen.

Figure 4.3 shows two different options for calculating the absolute value of an input value x. Depending on the value of x being either greater-than-or-equal, or less zero, x is returned unchanged or negated. In pure dataflow languages such a distinction would necessitate the modeling of parallel execution paths for every case. A terminating selector would then decide on one of the incoming values to be propagated as the overall result. This approach is shown in Figure 4.3(a). In contrast, an automaton is used to first check on the input value and, based on the result, evaluates the execution path defined in the respective state. In Figure 4.3(b) such an automaton is depicted. The former use of dataflow elements is clearly less intuitive to read and understand than the latter version using the automaton.

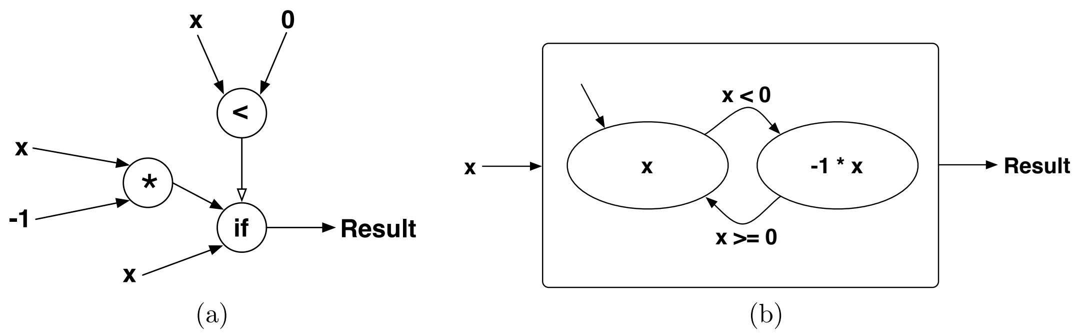


Figure 4.3: Comparison of control flow modeled without automaton in Figure (a) and using an automaton in Figure (b)

Better readability of a dataflow model can also be achieved by condensing a part of a dataflow graph into a single node, which is then called a *procedure*. We will also use the term *composite element* interchangeably with procedure. The number of input and output edges is identical to that of the replaced part of the graph. Hils refers to this technique as *procedural abstraction*. Using procedural abstraction repeatedly and building new networks of procedures, a refinement hierarchy can be established. Each individual hierarchy level of such a model is easier to grasp thanks to a reduced number of operators which are visible at once. COLA features the same abstraction concept, but the resulting compound elements are called *networks* rather than procedures, as we will show in Section 4.3. Using the abstraction concept, dataflow languages may also be used to specify a software architecture, not just basic algorithms.

Two important properties of dataflow languages make them well suited for the programming of parallel systems:

Freedom from side effects: Dataflow models are *functional models*, that is, they consist solely of operators transforming several input values into one or more output values, according to the function represented by the operator, as Ackermann points out in [2]. In dataflow languages, these operators are combined to form larger networks, by connecting the operators using edges. The edges depict the flow of data between the different operators. This explicit modeling of dataflow has two major advantages: first it prevents the existence of any complex side effects, since all inputs to an operator are modeled explicitly. It is not possible for an operator to access any data, if there is no connecting edge to the respective node. Second, the mathematical transparency arising form their functional nature, makes dataflow languages well suited for formal verification and automated model transformation.

Parallel modeling: Dataflow models are also *parallel models*. Hence they can be used to specify parallel executing algorithms, or parts thereof. Any guidelines for synchronization, and thus scheduling of systems derived from a dataflow model, are defined implicitly through data dependencies.

The absence of side effects is a basic requirement for efficient parallel programming, as described by Tesler and Enea in [120]. Together with the explicit parallelism apparent in dataflow networks, the derivation of a parallel implementation is made possible. The formal verifiability of a dataflow model is well suited to raise the model's quality regarding absence of implementation faults and the dependability of the resulting system. The dataflow approach therefore fits well two of the characteristics of reactive systems, namely parallelism and reliability.

In order to address the third property of reactive systems, that is the time constraints demanded for the system, dataflow modeling is not sufficient. This is why COLA — just like similar approaches — combines the concepts of dataflow languages with those of synchronous languages, as we will explain next.

4.2.2 Synchronous Languages

In synchronous languages time is seen as a succession of discrete steps, so called *ticks*. Any task modeled in such a language may be executed during each tick and the execution itself happens infinitely fast, that is, operations take no time. Analogously all communication between tasks is conducted in zero time. This idealized view allows the programmer to think of his programs to react instantly. Since inputs are only read and tasks only executed once during each tick, it is assured that the system reacts before another event occurs. To ensure the validness of this abstraction for a real system, the hypothesis has to be checked for a given system and environment. If the anticipated maximum frequency of events is fair and computing capacity of the hardware platform is sufficient, the actual mapping of logical ticks to a real time base can be chosen. This mapping has to ensure that each task is executed often enough to capture each change of the input values, and that the task operating on these values is finished before its subsequent invocation.

Synchronous languages have been proposed for the design of reactive systems by Benveniste and Berry in [16] and Halbwachs in [60] mainly for two reasons: first, the synchronous paradigm resembles the way design of control loop applications is typically carried out by control engineers. A set of inputs is cyclically read, which is called *sampling*, and interpreted by some microcontroller, which then outputs an according set of values. Second, tasks in such a system are implicitly synchronized, since the start of execution of each task is related to a global clock, which counts time in discrete steps.

From the above explanation the question arises if different tasks may be timed differently according to the assumed frequency for their input events. To this end most synchronous languages enable the definition of a clock for each task. The clocks of all tasks have to define ticks which are multiples of the basic clock of the system.

Another question is whether the real worst-case execution time of a task may be short enough to guarantee a termination of the task before its next invocation, especially when dealing with continuous mathematics used in control loop applications. This problem is addressed by using appropriate discrete algorithms for approximating the desired results. Since the use of loops might render the calculation of the worst-case execution time for a task impossible, algorithms should be designed to calculate rather an approximation during a repeated cyclic execution of the task, than during a single evaluation of the algorithm. Accordingly, the result of a synchronous model is typically a system based on a cyclically executed static schedule.

Several languages have been based on the synchronous paradigm, one of the first being Statecharts introduced by Harel in [64]. Other languages employed the same synchronous time assumption but were based on different programming principles like imperative semantics used in ESTEREL, which has been presented by Berry and Gonthier [22], or dataflow semantics which are used for LUSTRE proposed by Halbwachs et al. in [61].

4.2.3 Synchronous Dataflow Languages

The combination of the dataflow and synchronous approaches leads to the concept of synchronous dataflow languages. The synchronous paradigm gives a precise timing specification for the previously asynchronous dataflow approach. Thus, the point in time each operator is executed is defined and synchronized to all other operators. Just like LUSTRE, COLA is based on this combined approach.

In a synchronous dataflow language the evaluation of the operators contained in a dataflow model happens infinitely fast, that is, in logically zero time and, thus, all operations could be considered to be executed in parallel. Still, there is a causal dependency between the operators which is expressed by the dataflow edges. The dataflow semantics prevent an operator from being executed before all its inputs are available. Hence the assumption of zero time execution does not influence dataflow semantics in a negative way.

Another advantage of the synchronous approach, besides being easy to understand, relates to the execution time of composite elements — called *networks* in COLA. If the model assumption would assign an amount of time for the execution of each composite element, the designer would also have to specify the execution time for each of the operators — or procedures — contained in it. The sum of execution times of all these sub-components must then be equal to the execution time of the composite element. When dealing with lots of hierarchical refinement levels, this time specifications become tedious and error-prone to model. The zero time assumption of the synchronous approach renders this problem redundant, thus the developer can focus on the functional specification of his model. At the same time the design of a refinement hierarchy is alleviated which may be used to express a system architecture by summarizing sets of operators to larger composite elements.

As explained in the previous section, synchronous models consider time to be a sequence of discrete steps rather than a continuous value. Synchronous dataflow models are executed repeatedly according to these steps, seen as ticks of a global clock. If the model is partitioned into composite elements, it may be possible to assign a frequency for each composite element, depending on the language. Then the element is not executed each tick of the global clock, but instead, for example, every n-th tick. During each execution all the contained operators are executed in the order implied by the dataflow.

Now that we have outlined the basic principles of synchronous dataflow language, we will present our implementation of such a language, namely COLA.

4.3 The Component Language

The Component Language was invented for the design of distributed reactive systems, automotive systems in particular. Its advantage compared to other languages lies in its applicability throughout the entire automotive software development process. To this end it includes modeling constructs for specifying requirements as well as functional behavior and technical aspects of the in-car computing systems. The differentiation between these different views onto the intended system design is provided by the use of abstraction layers, each of which focuses on distinct aspects. The name Component Language is derived from the fact that each of these layers is modeled by — either software or hardware — components, which encapsulate a heap of other modeling constructs and together achieve the functionality of the system.

Another benefit of COLA lies in its foundation on formal semantics, which have been introduced in detail in [86]. This property forms the basis for automated inspection and transformation of a modeled system during model checking, system deployment, and model-level debugging.

COLA is intended solely for dealing with digital, that is, discrete data. Continuous values as they occur in the environment surrounding the system cannot be modeled. It is rather assumed that the execution platform conducts the necessary A/D conversion and provides the modeled software with digital inputs and similarly accepts digital output values. Further, COLA focuses on control algorithms and higher level logics. It is not suitable for the design of low-level software like device drivers. Because of the employed abstraction it simply lacks hardware details that would be necessary for low-level programming.

Before giving details about COLA's modeling constructs, we want to give an overview of the abstraction layers defined in COLA.

4.3.1 Abstraction Layers

As mentioned before, modeling in COLA is carried out using different *abstraction layers*. Each layer gives a different, simplified view onto the system, hiding those details which are unnecessary for the current modeling task. This concept makes the system easier to grasp for the respective developers. In addition it provides a separation of concerns by focusing, one after another, on requirements, functionality, or technical details like distribution and hardware figures. During the development process more and more details are captured in the model and a comprehensive model of the system is derived by finally combining information from all layers. This information is then available to the COLA tool-chain for model checking and code generation as well as system integration.

Analog to the scheme proposed by Broy et al. in [29] COLA features three abstraction layers, as are the *Feature Architecture*, the *Logical Architecture*, and the *Technical Architecture*. Their respective concern is as following:

Feature Architecture. The *Feature Architecture* (FA) is used to model the requirements regarding the automotive system from a user's point of view. Each feature in the model can be experienced by the customer in the resulting system. Further, intended and unintended feature interactions — as far as they can be anticipated — may be modeled explicitly and can be checked that way against the system's implementation which is specified in the Logical Architecture.

Logical Architecture. Using the *Logical Architecture* (LA) system designers model the functionality intended for the system under development. The Logical Architecture uses dataflow networks and mode automata to depict the according algorithms. Except for the available data sources and sinks, the Logical Architecture is independent of any hardware properties. Partitioning of the functional design into distributable units and allocation of these units to hardware elements is carried out using the Technical Architecture.

Technical Architecture. The *Technical Architecture* (TA) provides technical information related to the computation of functions, designed at the Logical Architecture, on an actual hardware platform. To this end it specifies an abstract representation of the hardware platform, capturing just its software relevant properties, and defines a distribution of the functions onto this platform. Besides, it comprehends the hardware's capabilities, hence providing the necessary figures for the calculation of allocation and scheduling schemes.

For the deployment approach in this thesis, the data comprised in the LA and TA are sufficient, as deployment does not take place before the requirements specified in the FA have been implemented in the LA and a hardware specification is available in the TA. Therefore we will describe the LA and TA in more detail, next, while an in-depth description of the feature structuring approach of the FA has been given by Rittmann in [111].

4.3.2 Feature Architecture

The purpose of the FA is to structure the requirements for a system. To this end, the requirements are brought into a tree-like hierarchy. The result is a so-called feature tree, which shows the composition of complex features using a number of simpler sub-features. In addition to this hierarchical composition, the feature tree also includes feature interactions, which indicate desired or undesired interactions of the contained features. These feature interactions can be specified either using COLA language constructs of the LA or by temporal-logical formulae expressed in the *Smart Assertion Language for Temporal Logic* (SALT), which has been introduced by Bauer et al. in [15].

Further, the maximum deadline for the execution of a feature may be specified to define an upper limit for its timely execution. These timing figures are transferred to the subsequent abstraction layers to derive timing information for the respective implementation, which may finally be used to calculate a valid scheduling plan.

Figure 4.4 shows an example feature tree from the COLA editor. The top node of the tree represents the automobile, its children are different groups of features related to comfort, safety, entertainment, etc., followed by their respective implementing services. The vertical connections arise from the specification of feature interactions and make the tree appear as a graph structure. Still, the features are arranged in a strictly hierarchical manner.

The leaves of the tree represent different functions required for the target system. These leaves are therefore used to derive an initial architecture of software

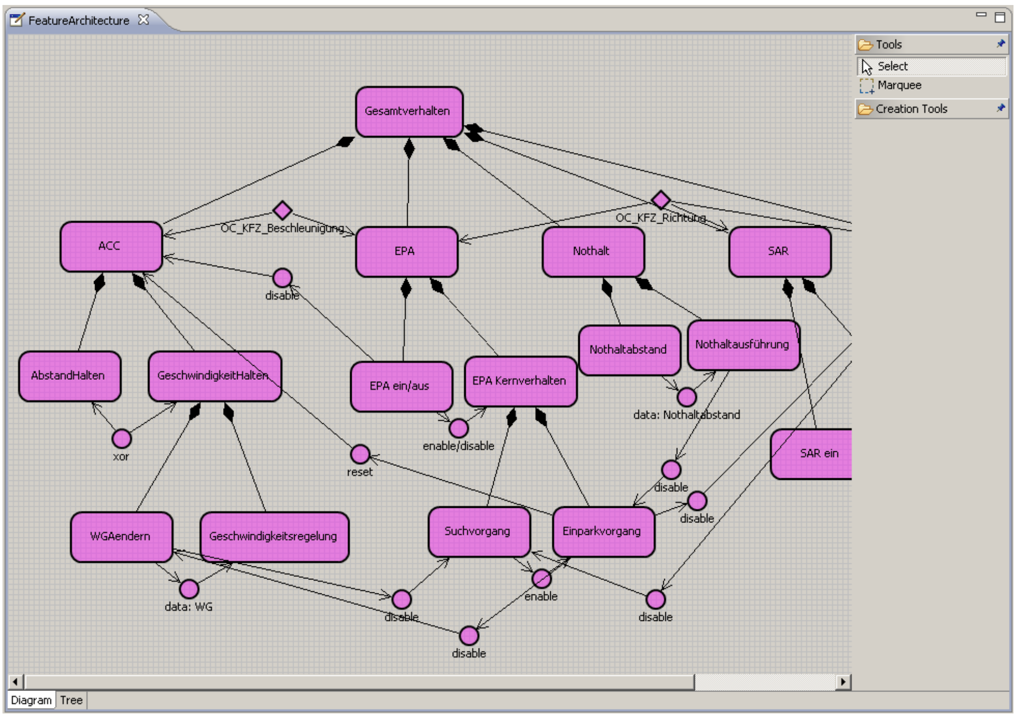


Figure 4.4: The Feature Architecture

components for the LA. Modelers use this architecture as a starting point and fill the software components with functionality using constructs of the LA.

A more detailed description of the concepts of the FA can be found in [111].

4.3.3 Logical Architecture

The LA is used to specify the functional behavior of a system design. Its constructs are used to model an overall implementation of the system, that is, it is not limited to a subset of tasks or single computing nodes. The system is rather designed as a whole, without taking any partitioning or distribution onto a specific hardware platform into account. Those jobs are done later on, using the TA. The LA is based on the concept of dataflow, hence the diagrams of the LA consist of procedures and dataflow edges between those operators. The operators are named *units* in COLA, while the edges are called *channels*.

Definition 4.1. *A* ***unit*** *is the generic type for operations in a* COLA *diagram. Each unit exhibits a set of input and output connectors called ports.*

Units as a generic type are the placeholders for different COLA elements. Figure 4.5 shows the (simplified) meta-model of the COLA language. We will explain

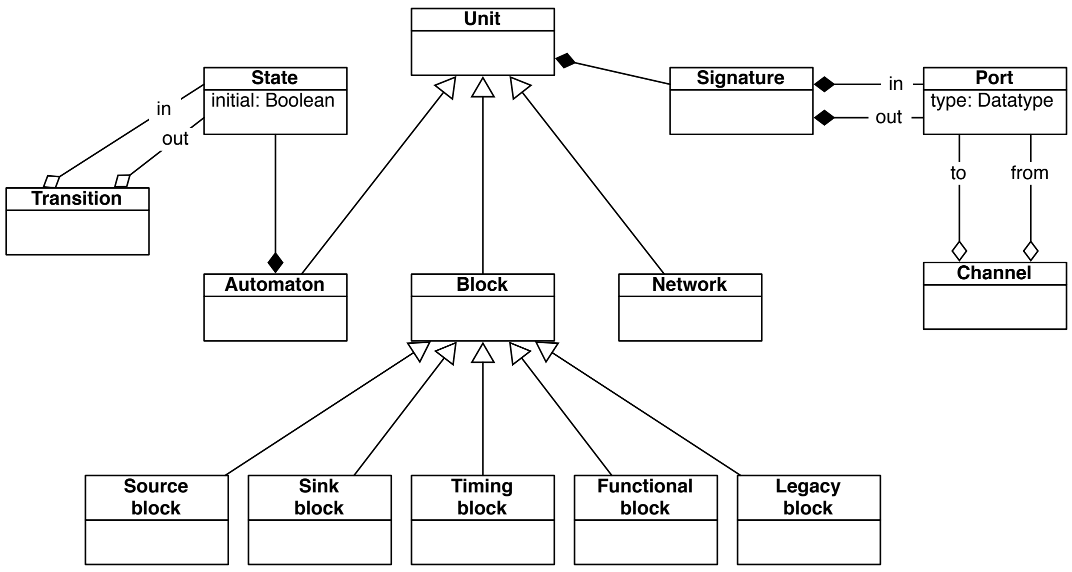


Figure 4.5: The COLA meta-model

the COLA modeling elements shown in the figure in the remainder of this section. As can be seen in Figure 4.5, the real implementation of a unit may be a network, an automaton, or a block, each of which we will explain in the following.

Definition 4.2. ***Ports*** *define the input and output points of a unit. Each port is given a specific type and only ports of compatible type may be connected to it. The set of all input and output ports of a unit is called signature.*

Each unit has one or more ports, where every one may be connected to the respective ports of other units, thus representing an exchange of data between the two units. In the graphical syntax ports are represented as small triangles on the borderline of a unit. Input ports point inwards, while output ports point out of the respective unis. Connections between ports follow a $1 : n$ scheme, that is, each output port may be connected to several input ports, but each input port excepts only data from one output port. The COLA semantics require each input port to be connected to some port of another unit, since the input of the unit would be undefined otherwise. Output ports, however, do not have to be connected to a subsequent input port. If they are not, their results are simply dropped. The connections between different ports are depicted by channels.

Definition 4.3. *A* ***channel*** *depicts a 1:n dataflow connection between the output port of a unit and the input port(s) of (a) subsequent unit(s).*

COLA semantics prohibit channels connecting an output port of a unit to any of the input ports of the same unit. Such a channel would depict a feedback loop with potentially unknown runtime, which is not acceptable for a real-time system with stringent deadlines. Rather, the loop has to contain a delay.

Definition 4.4. *A* ***delay*** *is a* COLA *timing operator with exactly one input and one output port. Whenever a value is provided at the input port, it stores that value and in turn emits the value provided during the last execution at the output port.*

Delays are buffers which store a given value for exactly one execution cycle of the model. To guarantee a defined output during the first execution of the model, delays must always have a default value which is defined by the modeler at design time.

To build larger sets of operations, networks are used in COLA. A network comprises a set of units, that is again networks, automata, or blocks, and defines data exchange between those units.

Definition 4.5. *A* COLA ***network*** *is a unit which contains a set of other units — referred to as* sub-units *— that are interconnected by channels. The sub-units may use the network's input ports and have to write to its output ports.*

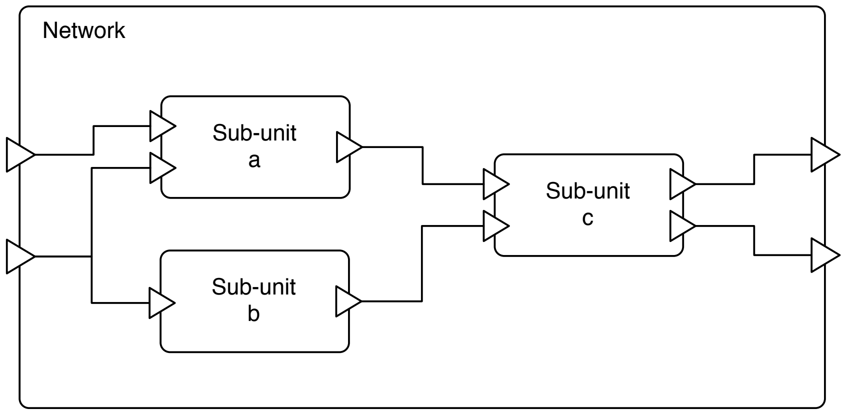


Figure 4.6: A COLA network

Figure 4.6 shows an example for a COLA network containing three sub-units. The network is depicted by the outer box, enclosing all the sub-units of the network. The triangles pointing into the box at the left side display input ports, while the triangles at the right hand side point outwards to illustrate output ports. Data are emitted into the network via the input ports, hence the sub-units may be connected to any of the input ports using channels. COLA does not require all input ports to be used, that is input ports may remain unconnected from the contained sub-units. On the contrary, all output ports of the network have to be provided with data. Thus each output port of the network must be connected to exactly one of the sub-units' output ports.

As can be seen in Figure 4.6 the direction of the ports together with their connecting channels induce a causal order regarding the evaluation of the sub-units. Starting at the input ports, data are available for `Sub-unit a` and `Sub-unit b` at the same time. Thus these two units may be evaluated in any order, or even in parallel. `Sub-unit c` in contrast may not be evaluated before the execution of sub-units `a` and `b` is finished, because it is dependent on their outputs. Finally, `Sub-unit c` writes the output values of the parent network.

Using the concept of networks it is possible to define a hierarchy of refinement steps by inserting networks as the sub-units of another network. Before a COLA model can be transformed into code, an ending point of the hierarchy, which is the most fine-grained hierarchy level, has to be defined. This lower ending point of the refinement hierarchy is embodied by blocks.

Definition 4.6. COLA ***blocks*** *form the atomic logical and arithmetic operations of a* COLA *model. They are the most fine-grained elements and my not be refined further.*

The meta-model in Figure 4.5 indicates, that the abstract element block is realized either by a functional block, a timing block, a source or sink block, or a legacy block. As described before, delays are timing blocks in the COLA language. Functional blocks represent basic arithmetic and Boolean operators. Source or sink blocks form the connecting points to the hardware platform. Each source block is related to a data source, that is a sensor, of the hardware platform, while sinks are the model representation of actuators.

Definition 4.7. ***Sources*** *and* ***sinks*** *model the links between a functional model and devices of the target platform. Sources are provide input to the model by representing a sensor of the platform, while sinks define the possibility to output data to the environment, using an actuator.*

Figure 4.7 shows an example network consisting solely of blocks. The shown network implements the Pythagorean equation $\mathtt{a}^2 + \mathtt{b}^2 = \mathtt{c}^2$, taking `a` and `b` as input values and returning $\mathtt{c}^2$.

So far, all presented modeling elements adhere to the dataflow concept. But COLA also features a modeling construct to represent control flow, so-called automata. Just as well as control flow defines a way to differentiate between one execution path or the other, COLA automata enable a decision between different behaviors according to some condition.

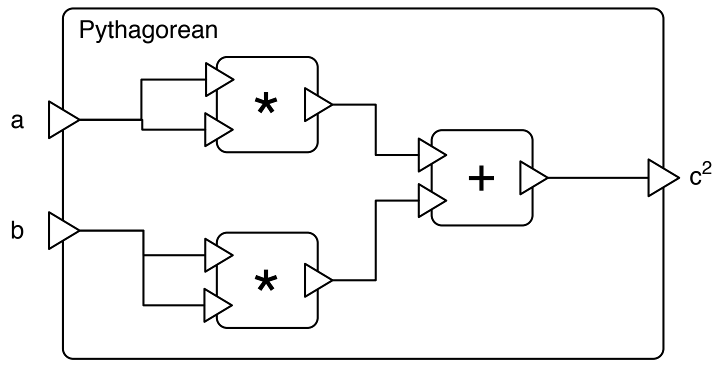


Figure 4.7: A network of blocks

Definition 4.8. *A* COLA ***automaton*** *holds a set of states of which exactly one is active at any time. The transitions are controlled by guards which decide if a transition from one state to the next is taken or not.*

Figure 4.5 shows that the states contained in an automaton are again implemented by units. Each state of a COLA automaton is implemented by either a network, another automaton, or a block. Analog to networks this allows for hierarchical refinement. The COLA semantics require the states, or their related implementing units, to have the same signature as the automaton itself. This way each state has the same number and types of ports as the automaton. During execution exactly one of the states is active. At the beginning of each invocation of the automaton, the outgoing transitions of the currently active state are considered. The transitions' preconditions are referred to as guards in COLA.

Definition 4.9. ***Guards*** *are modeled by networks which consist solely of blocks and emit a single Boolean result value. Delay blocks may not be used inside a guard network. If the guard of any of the transitions evaluates to true, the transition is taken and the behavior defined for the respective target state of the transition is subsequently executed. If all guards evaluate to false, the already active state's behavior is executed.*

The guards' input values are identical to those of the automaton. Thus the guards may use all or a subset of these values for their processing. COLA semantics require automata to be deterministic, that is only one transition may be taken at any time. The model checking of a COLA model includes the necessary tests to make sure guards do not define non-deterministic automata.

An example automaton is shown in Figure 4.8. Depending on the last active state, the outgoing transition of that state is checked. Assuming that `State 1` was

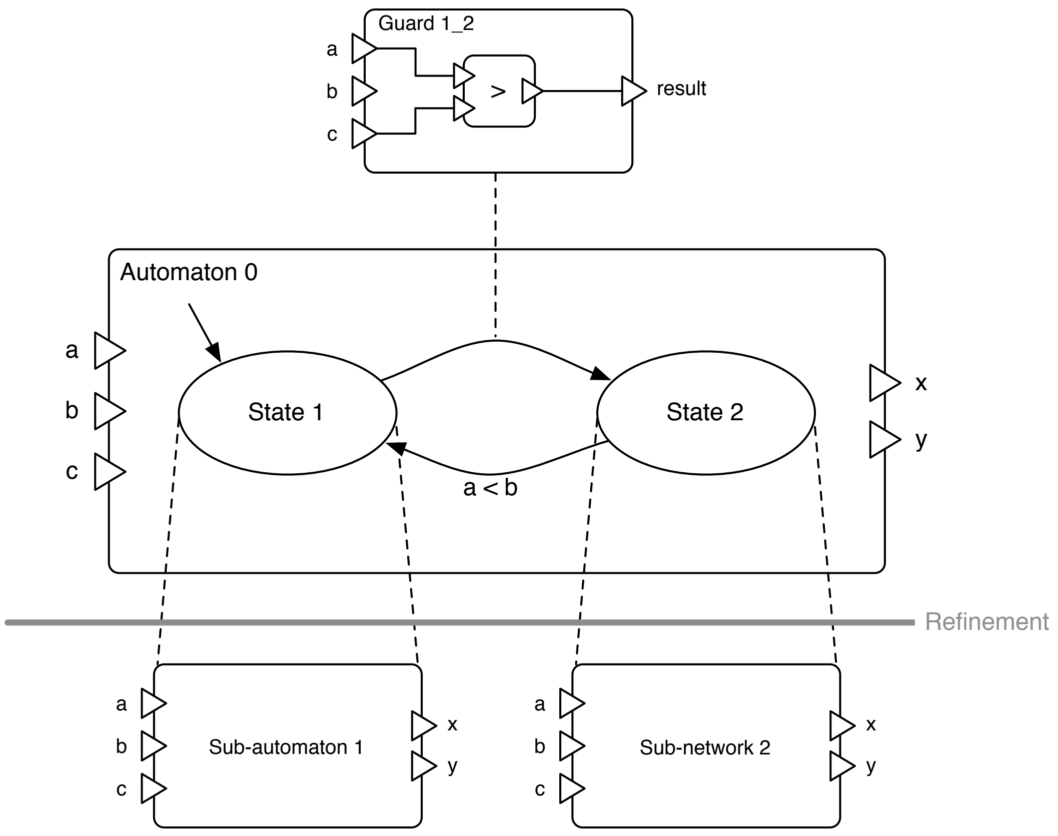


Figure 4.8: A COLA automaton

the last active state, the transition indicated at the top of the figure has to be evaluated. For clarification the guard of the transition from `State 1` to `State 2` is shown in graphical COLA notation, while the second guard is depicted textually. The guard from `State 1` to `State 2` emits `true` as its result, if $a > c$. In that case the transition is taken and the implementation of `State 2` is executed subsequently, which would be `Sub-network 2` in this example. `State 1` is executed, otherwise, implemented by `Sub-automaton 1`. Please note that the signatures of `Automaton 0` and its sub-units are identical, while the guards share the same input ports, but have only a single Boolean output, as mentioned before.

Definition 4.10. *In* COLA *automata and delays are* ***stateful units****. A stateful unit retains its internal state, that is, its value between cyclic invocations of the model.*

Most modeling constructs of COLA are stateless, that means they do not retain any values. The only exceptions are delays and automata. As described, delays store the current input value during each invocation. Similarly, automata have to buffer the actually selected state until their next invocation. For all stateful units default values have to be specified to assure a defined behavior of the system. It is the modelers responsibility to provide these values.

This completes the informal description of the constructs available in the LA. The features of COLA presented here are limited to those information necessary to understand the deployment approach. For more details about the COLA language, please refer to [86].

4.3.4 Technical Architecture

The presented approach is targeted at distributed hard real-time systems. Regarding the automotive domain, such a system consists of a multitude of processing units each of which is equipped with one or more processors, memory, a communication interface, and devices to interact with the environment. We refer to processing units with their dedicated hardware as (computing) nodes of the real-time computer system, or ECUs with respect to the automotive domain.

As it is our goal to generate the code for the system's functionality as well as the platform configuration for scheduling and communication, a mapping of tasks to executing hardware nodes is necessary. To this end COLA comprises an abstraction layer denoted as Technical Architecture. In order to distinguish between the software components, the ECUs, and the mapping between those, the Technical Architecture is divided into three different views. These views decrease the number of modeling elements visible according to the current modeling task. Their respective names are *Cluster Architecture*, *Hardware Topology*, and *Allocation*. We will introduce each of these views in the following.

Cluster Architecture. It is the task of the Cluster Architecture to define a partitioning of the highly connected LA into distributable software components. These software components are later on, during deployment, transformed into code and become tasks in the executable system. The model representation of a task is a cluster.

Definition 4.11. ***Clusters*** *combine a set of units from the* COLA *LA model into a distributable entity. A cluster becomes a task in the executable system and may only be allocated to a single ECU and executed as a whole.*

A cluster is an atomic item from the deployment's point of view. The decision which units of the LA to put into one cluster can be made either manually or computed automatically according to some heuristics, as has been mentioned in Section 4.1. For a valid Cluster Architecture, all units have to be contained in at least one cluster after partitioning is finished. Two types of clusters can be distinguished, the *working cluster* and the *mode cluster*.

Definition 4.12. *A* ***working cluster*** *comprises a unit and all of its sub-units. Each working cluster is refined down to the level of functional blocks. All of the contained sub-units are executed according to the dataflow semantics. Working cluster must not overlap each other.*

To specify a working cluster, a parent unit is selected from the LA. This unit together with all its sub-units is then considered a distributable entity and will be generated into a piece of code during deployment. In addition to working clusters, mode clusters may be defined. Mode clusters must consist solely of automata, the states of which are implemented either by further mode clusters or by working clusters.

Definition 4.13. *A* ***mode cluster*** *groups several clusters into distinct sets which are activated alternately. To this end a mode cluster consists solely of automata which are not refined down to the block level. Instead, each state is again implemented by some cluster. The automata's active states specify, which clusters to execute.*

Figure 4.9 gives an example partitioning of a model from the LA into working and mode clusters. The shown subsystem changes its behavior depending on the current ignition state. The top-level automaton features two states named `Ignition on` and `Ignition off`. Inside the ignition on state, a further distinction is made whether the engine is running or not, using another automaton. Each of the automata is defined to be a mode cluster, shown in dark gray. The remaining units shown in the figure, in contrast, implement the afore mentioned operating modes. Thus, they are defined to be working clusters, indicated by their light gray color. The figure illustrates the fact, that mode clusters may contain other clusters, while working clusters may not.

Just like working clusters, mode clusters are generated into code and executed as tasks on the target platform. They are used to modify the set of activated tasks during system execution. These different sets of tasks are referred to as operating modes.

Definition 4.14. *An* ***operating mode*** *defines the set of clusters — which may be a combination of some mode clusters as well as some working clusters — to be executed in a certain state of the system. Thus, it is possible to define a hierarchy of operating modes.*

The definition of operating modes by means of automata has been proposed by Harel in his work about Statecharts [64]. Their use in safety critical systems has

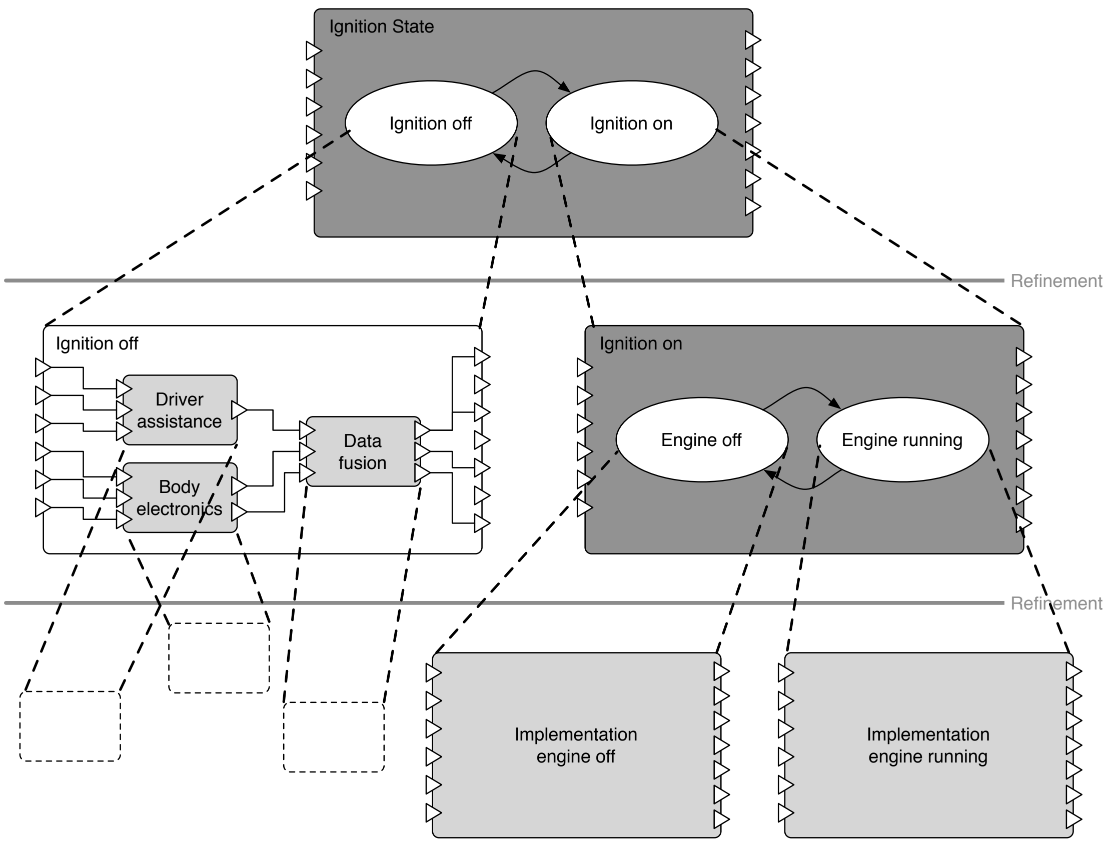


Figure 4.9: A possible clustering scheme

been first proposed by Maraninchi and Rémond in [97]. Examples for operating modes in an actual car would be parking, driving, ignition off, ignition on, dry conditions, wet conditions, and the likes. According to the actual mode, only the respective clusters are executed, avoiding the evaluation of currently unneeded clusters in the resulting system. Hence, a system which is based on operating modes can be designed more resource efficient, that is, it consumes less processor power and, accordingly, less energy. At the same time a less capable hardware platform might be sufficient for executing such a system, reducing the overall cost.

Hardware Topology. To complement the Cluster Architecture with a model of the target system, COLA provides the Hardware Topology as a part of the TA. The Hardware Topology contains various information about the target platform, including ECUs, sensors and actuators, employed communication systems, and logical topology together with their respective characteristics. The degree of detail is limited to those characteristics necessary for an automated deployment.

As can be seen in Figure 4.10, the hardware topology consists of *processing nodes*, *sensors*, *actuators*, *buses*, and *connections* between those elements. We will explain

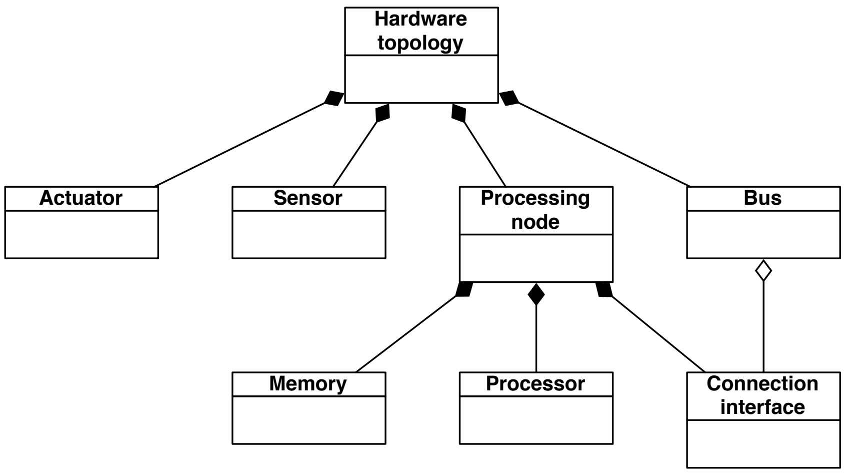


Figure 4.10: The COLA hardware topology

these elements in this paragraph.

Definition 4.15. *A **processing node** is the model representation of an ECU. The attributes given for the processing node provide a description of the ECU's hardware characteristics, which serve as a basis for the deployment of a* COLA *model.*

Essentially, ECUs are embedded computing devices consisting of a microprocessor, some memory, and eventually some special hardware necessary for the desired function. Since our hardware model serves as a guideline for allocating software tasks, we are interested in the nodes' characteristics influencing the computation of a task. These characteristics are modeled as attributes of the processing node in the Hardware Topology.

The nodes' performance is expressed by the number of *processing cycles* per millisecond and amount of memory available. The number of processing cycles can then be matched to the worst-case execution time (WCET) for every task which might be executed on a node. If enough processing cycles are left on the node, it is valid to allocate the task to that node.

Concerning the amount of memory we have to distinguish between memory for program code and program execution. The node has to carry therefore information about its *ROM* and *RAM* sizes. The memory's speed is not looked at separately because it is already included in the processing performance evaluation of the node.

Regarding the schedulability of a set of tasks, additional node specific factors have to be considered. The local operating system will consume some processing power for dispatching, memory management, interrupt handling, etc. This *OS overhead* has to be measured in a realistic simulation environment. The node's processing speed value has to be decremented by this amount before allocating the tasks.

Additional tolerances concerning the scheduling of a distributed application on several nodes have to be considered, since even in a time-triggered system there is always some clock drift between several nodes. Thus the *drift rate* of each node is of interest, which depends on the quality of the hardware timer used.

By using these figures, it is possible to derive a valid set of tasks which may be allocated to a processing node without demanding more than the available resources. For a valid allocation of the overall system, however, the deadlines of applications specified in the FA of the COLA model have to be considered. These applications may be implemented by a set of tasks distributed over several processing nodes. To facilitate the necessary exchange of data, the processing nodes are interconnected by a communication system. The key figures of this communication system are expressed using *connection interfaces* and *buses* in the Hardware Topology.

Definition 4.16. ***Connection interfaces*** *define the interface between a processing node and a bus system. The interface is annotated with technical information about addressing, timing, and data buffering which are necessary for platform configuration.*

The connection interface encapsulates technical specifications about the access of a processing node to a bus. This includes the definition of the bus interface as well as the according *bus id* for each interface the node provides. The bus id is used for addressing messages from and to the node by the bus protocol.

Further the values for *com buffer size* and *com delay* have to be given for each interface. These indicate the amount of RAM and processing time to reserve for the communication protocol in question. These data are taken into account during schedulability analysis of a possible allocation. In addition, the specifications of the employed bus system are needed.

Definition 4.17. *A* ***bus*** *is the model representation of a physical communication system between ECUs. To calculate the end-to-end communication time for applications using the communication system, its characteristics are modeled as attributes of the bus.*

The characteristics of buses used in a distributed system influence severely the performance of the system. The target architecture shall provide a bus which uses a time-triggered protocol for communication. First the protocol's *net bandwidth* and *maximum packet size* are of interest. These data are used in combination with the *slot length* of the communication slots to calculate the communication schedule. Regarding the timing accuracy of the system, the *synchronisation rate* and *jitter* of the buses have to be given. The synchronisation service is usually provided by the bus protocol, so the synchronisation rate depends on the protocol used. The jitter induced by the bus decreases the accuracy and has to be taken into consideration.

For interaction with the physical environment of the car, automotive system employ a multitude of sensors and actuators. These sensors and actuators form the counterpart of sources and sinks in the LA. Thus for each source or sink of the LA, an according sensor or actuator should be present in the Hardware Topology.

Definition 4.18. *In* COLA *a* ***sensor*** *is the equivalent of a hardware sensor providing discrete (digital) data for the system. The model representation of the sensor provides information about its hardware capabilities as well as a definition to which processing node the sensor is connected.*

We assume sensors to be connected to an A/D converter of the ECU. This is why the output of a sensor is always a discrete value in the model. When dealing with sensors, we are interested in the *range of values* they produce as well as the *resolution* of these values. These information can be used to allocate buffers of appropriate type during deployment and to check the characteristics given in the software model. Further the *sampling rate* of the sensor is of interest. It indicates the length of the polling interval at which a trigger task has to be invoked.

Definition 4.19. *In the* COLA *Hardware Architecture an* ***actuator*** *is a sink for discrete data. Actuators are, like sensors, connected to a certain processing node.*

Similar to sensors we assume for our model that actuators are directly connected to a D/A converter of some processing node. For the invocation of an actuator we also need the accepted *range of values* and, accordingly, the *resolution* it accepts. The actuator may be written to as often as indicated by its *sampling rate*. The deployment tools for COLA use the information about sensors and actuators to assure their invocation during execution and communicate the according values using the bus system. According to our approach the devices will be interfaced by a middleware, the configuration of which we will describe in Section 5.7.

Allocation. To provide a relation between Cluster Architecture and Hardware Topology, an allocation from clusters and sources/sinks to hardware components is defined.

Definition 4.20. *The mapping of modeling constructs to hardware components they are executed on is denoted as* ***allocation****.*

Mode and working clusters may be allocated to any processing node, if the node has sufficient resources left and the clusters do not require any specific hardware. In contrast, for sources and sinks there is exactly one valid place to be allocated, namely the sensor or actuator they refer to. An allocation is valid if all clusters and sources/sinks are allocated to an appropriate element of the Hardware Topology without overcharging the resources available. However, this does not yet guarantee a functional system, since the chosen allocation must also be schedulable. It is the job of the scheduling tool to check for a valid schedule and, if none is found, a different allocation has to be chosen and checked for schedulability again. We will provide more details about deployment in general, as well as allocation and schedulability in particular, in Chapter 5.

4.3.5 Timing Assumption

The COLA language adheres to the concept of perfect synchrony, which has been introduced in Section 4.2. It is assumed, that the entire system modeled in the LA is executed each tick of a discrete global clock. During each tick all units are evaluated in (logically) zero time. Just the precedence of units implied by the dataflow has to be considered regarding the evaluation order. In the model, the global clock is incremented at each tick. The correlation of ticks to a real-world wall clock is fixed in the scheduling step during deployment.

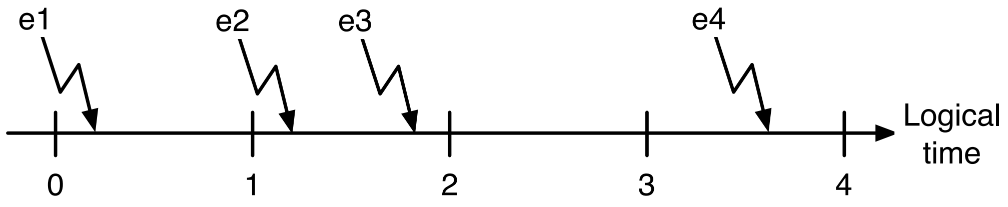


Figure 4.11: Logical timing of COLA models

Figure 4.11 shows an excerpt from a possible model timing. At each tick of the logical clock, illustrated by markings on the number line , the entire model is evaluated. Events generated by the environment, indicated by events `e1` through `e4` in the figure, may reach the system at any point in time. The inputs given by these events are not considered by the system before its next evaluation. Hence, `e1` is interpreted at tick `1` and `e4` is interpreted at tick `4`. For ticks `2` and `3` it has to be noted that COLA is based on a last-value semantics. If `e2` and `e3` are events of the same type, and thus are captured by the same sensor, `e3` overwrites `e2`. Since there is no evaluation between the two, the value of `e2` is simply lost. Similar, at tick `3`, the value given by `e3` is used again, since no new input has been received.

To specify the actual timing requirements of an application, the COLA FA is employed. For any node of the FA graph, a deadline for the execution of the respective functionality may be defined. After the functionality has been implemented at the LA, the WCET for all clusters involved by the application may be calculated, as we will describe in Chapter 5. Adding these WCETs to the respective communication and scheduling delays enables the scheduling tool, which will also be described in the deployment chapter, to derive a valid schedule.

Of course the question arises, how to choose the cycle length according to wall clock time for one tick of the model time. The scheduling algorithm tries to calculate a valid solution which guarantees, that first, the cycle is long enough to allow all clusters to be executed, and second, the cycle is short enough to meet the required deadlines for all applications. If the model may not be scheduled because of a lack of hardware resources or a model which poses excessive timing constraints, the scheduling tool will output an error and require the developer to refactor the model. This refactoring might either consist of a different allocation or, if this does not solve the problem, a different clustering scheme.

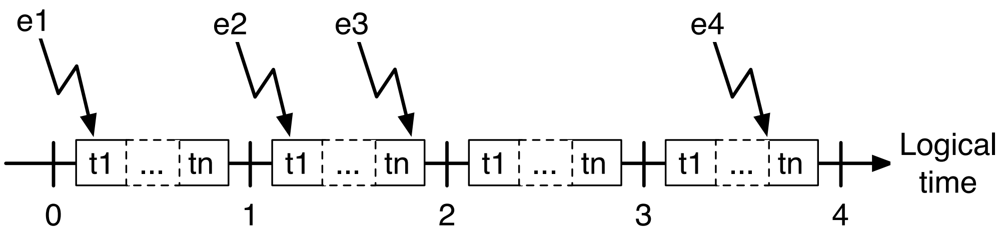


Figure 4.12: Relating task execution times to logical time

Figure 4.12 depicts the mapping of ticks of the logical time to actual task execution times. As can be seen in the figure, between any two ticks tasks `t1` through `tn` are executed, which implement the modeled system. Equivalent to Figure 4.11, only the events available before the execution of `t1` are considered. Events occurring during the execution of the tasks are stored until the next scheduling cycle and may be overwritten, as it is the case for `e2`. If this behavior is undesirable, the actual time between the ticks has to be chosen shorter, which in turn requires more powerful hardware to enable the execution of all tasks.

Obviously, it would be desirable to specify units in COLA, which are executed more often than others. Partly, this was taken care of by defining a last-value semantics for all COLA ports. Each output port always retains this way its last value which may be read multiple times by the connected input port(s). In similar fashion the value will be overwritten by a newer result, even if it has not been read yet. A COLA extension regarding different execution frequencies of units which would enable units to be only executed every "n-th" tick is still missing and will be subject to future work.

We will describe the scheduling scheme for mapping the synchronous time model onto our time-triggered platform in more detail in Chapter 5.

4.4 Tool Support for COLA

For the evaluation of the approach described in Section 4.1, a prototypical tool-chain based on COLA has been implemented. At the core of the tool-chain is the COLA model editor which is based on the Eclipse platform[1]. Eclipse provides a mature platform for the implementation of textual and graphical editors, and allows the addition of functionality by means of plug-ins. The COLA editor can be used to design FA, LA, and TA of a system and to control all the plug-ins operating on the system design. The model editor implements some basic model checks, which include compatible types of connected ports, a check for the presence of delays in feedback loops, etc. Furthermore, all automata of the model may be checked for determinism.

If a model of the LA is given, model-level simulation may be carried out using the COLA simulator, as explained in detail by Herrmannsdoerfer et al. in [71]. Figure 4.13 shows the graphical interface of the simulator. It is integrated into the COLA editor and equips the model under simulation with the respective simulation data. As can be seen in the figure, the simulator interface provides several panels for interaction. To specify the inputs for a unit under simulation, the runtime configuration panel can be used. It enables the user to define a value for each input of the unit. Using the control panel the developer may start, pause, resume, and stop the simulation at any desired point. The simulator reads the given inputs and calculates results according to the unit's COLA semantics. The resulting valuations of all ports and the active automaton states contained in the unit are displayed textually in the runtime configuration panel, or annotated in the model next to the respective model elements.

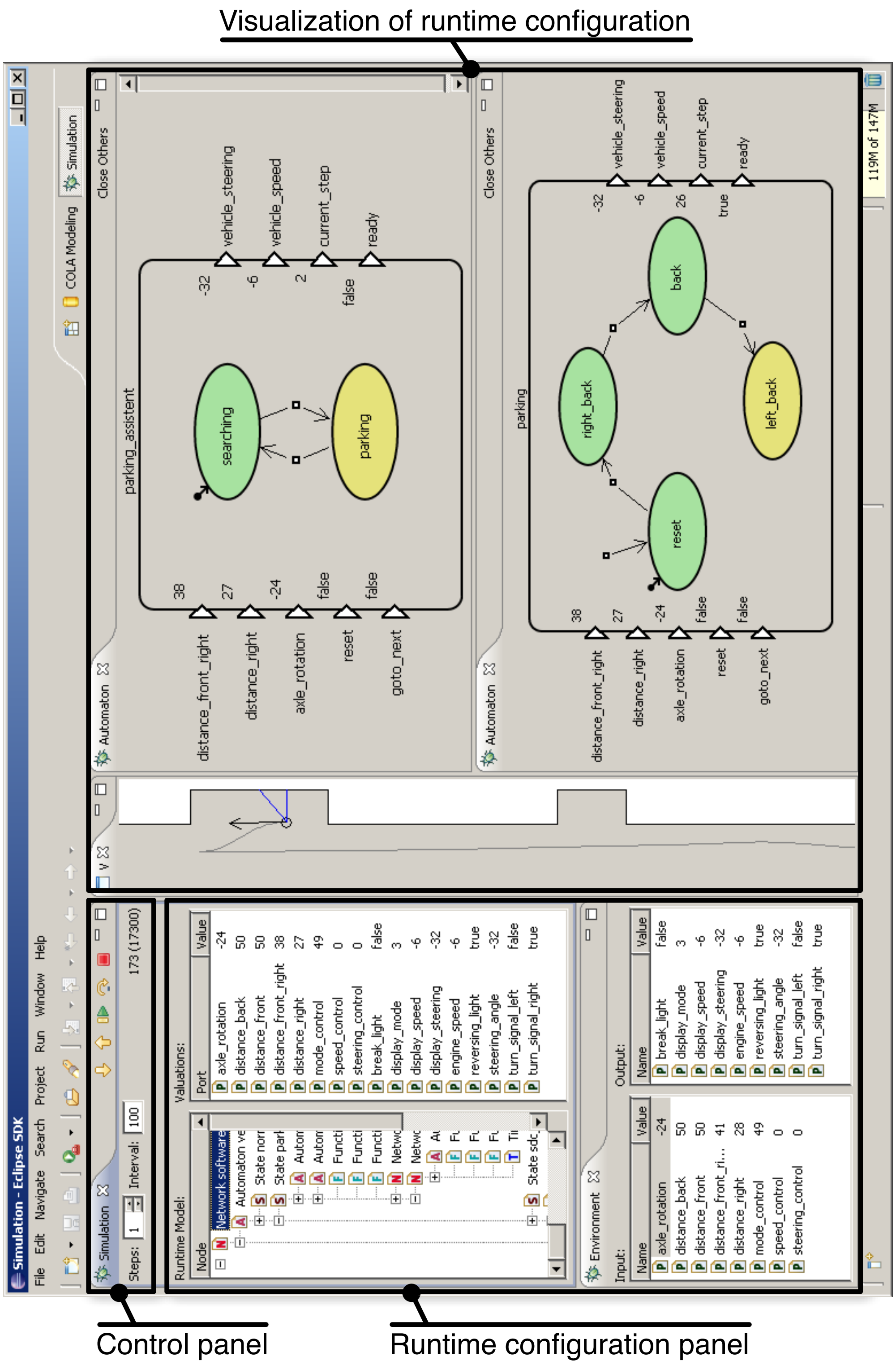
The simulator forms also the basis for our model-level debugging approach, which we will present in Section 6.1.

In addition to modeling and simulation, a set of plug-ins for the deployment of COLA models has been implemented. These plug-ins are used to perform the system partitioning, code generation, resource estimation, allocation, scheduling, and system configuration steps, as described in Section 4.1. All the deployment tools may also be started from the COLA editor. The mode of operation of these plug-ins will be explained in detail in the next chapter.

4.5 Related Work

Considering the growing importance of embedded real-time systems, it does not come as a surprise that a multitude of concepts is available for their model-driven development. We have selected some of the best-known approaches and will outline

[1]http://www.eclipse.org



Valuations:

Port	Value
axle_rotation	-24
distance_back	50
distance_front	50
distance_front_right	38
distance_right	27
mode_control	49
speed_control	0
steering_control	0
break_light	false
display_mode	3
display_speed	-6
display_steering	-32
engine_speed	-6
reversing_light	true
steering_angle	-32
turn_signal_left	false
turn_signal_right	true

Input:

Name	Value
axle_rotation	-24
distance_back	50
distance_front	50
distance_front_ri...	41
distance_right	28
mode_control	49
speed_control	0
steering_control	0

Output:

Name	Value
break_light	false
display_mode	3
display_speed	-6
display_steering	-32
engine_speed	-6
reversing_light	true
steering_angle	-32
turn_signal_left	false
turn_signal_right	true

Figure 4.13: The COLA simulator

their differences to our approach in this section. The section consists of three parts, according to the different nature of these concepts.

First there are solutions focusing on modeling the architecture of a distributed real-time system. Second we will present concepts being mainly intended for modeling the behavior of those systems. Third an introduction to platform concepts for the execution of distributed real-time systems is given. These platforms also provide a means of model-driven development since they are configured offline using some model of the actual application. At runtime the respective platform monitors the applications and prevents them from occupying more resources than permitted.

4.5.1 Concepts for Modeling Architecture

Architecture modeling is important, in particular, for distributed embedded systems. Here not only the internal software architecture has to be covered, but also the interaction of tasks on different nodes of the system has to be considered. To this end architecture modeling for embedded systems often comprises a means of modeling the hardware platform. Using this information the physical distribution of tasks may be modeled. We have already given a short introduction to AUTOSAR in Section 3.3 which enables the design of automotive system architectures. In the following, two alternative concepts are described, namely MARTE and EAST-ADL.

Over the years, the UML [114] has become one of the best-known and widely used concepts for software modeling. It provides a lot of different diagram types for designing various aspects of the system in question. A common critique regarding its use for modeling embedded real-time systems was its inability to model time at the desired granularity.

Therefore a UML profile has been defined which is called *Modeling and Analysis of Real-Time and Embedded Systems* (MARTE). MARTE extends the UML with constructs which allow the modeling of specific aspects of those systems. To this end, MARTE is partitioned into three main packages. The *foundations package* contains constructs for modeling non-functional properties of a system, which include timing and allocation of tasks. The *design package* is employed to model hardware and software resources. While the *analysis package* is used to analyze schedulability and performance of the system [41, 44].

However, the UML lacks formal semantics, as pointed out by Amálio et al. in [4] and France et al. in [47]. Without clear semantics available, the results of a timing analysis based on MARTE are subject to the semantics of the respective tool. In comparison to COLA, MARTE is not aimed at modeling the functionality of a system, but is limited to its structural aspects. Due to these disadvantages, MARTE is not suitable for generating a complete system from the model due to a lack of information. Rather it may be used to produce code stubs which may then be refined manually.

The *Electronics Architecture and Software Technology - Architecture Description*

Language (EAST-ADL) [37] enables modeling of combined software and hardware architectures at different levels of abstraction. Like MARTE, it is based on the UML, and features extensions for the modeling of automotive systems. EAST-ADL is also intended to model the structural aspects of a system. Requirements for the systems timing and behavior may also be specified, but are not detailed enough to enable code generation therefrom. As a solution, EAST-ADL provides a concept for referencing existing application code in order to specify the exact behavior of a software component. Another possibility is a reference to a MATLAB/Simulink model. We will introduce MATLAB/Simulink in the next section. Compared to COLA, the use of different modeling concepts in EAST-ADL contradicts the idea of an integrated tool-chain. As a result incompatibilities may be encountered at the contact points of the different languages.

4.5.2 Concepts for Modeling Behavior

Besides architecture modeling, a number of concepts and tools for behavioral modeling are also available. They focus on the functional design of embedded real-time systems by means of models. For this purpose, dataflow representations are very common.

Lustre [61] is an example for such a language. It shares with COLA the concept of synchronous dataflow. Further, it is also based on formal semantics, making it suitable for model checking. In contrast to COLA, Lustre is limited to the functional design of a system. It does not provide any constructs for modeling requirements or the hardware platform. So it has to be used in combination with other tools to cover an entire development process. Therefore it cannot be said to be a seamless modeling approach, which was a main reason for the invention of COLA. Seamless development allows for traceability of errors throughout the entire development process and thus enables shorter development cycles and improved system quality.

Lustre features only textual syntax. The *SCADE* tool [1] provides a graphical editor for the Lustre language. Further, target code may be generated using SCADE. For SCADE, which is based on Lustre, a deployment concept for distributed embedded systems has been presented in [34]. Compared to COLA, this approach lacks two key concepts. First of all, without information about the platform available, it does not offer the automatic deployment for a distributed platform. While the approach does facilitate the generation of schedules, it relies a given allocation of tasks to computing nodes as its input. Automatic computation of allocations is not available. Second, the approach lacks the generation of operating modes, although Lustre supports mode automata.

The previously mentioned *ASCET-SD* and *MATLAB/Simulink* are two more well-known tools for the development of automotive systems. While ASCET-SD is intended specifically for designing automotive software, Simulink is targeted more general at embedded systems. Both tools provide graphical modeling similar to COLA. However they do not implement synchronous languages. Rather, the modeled systems are triggered by events. To this end special event sources are included in the system design. Hence, synchronization between different parts of the system has to be resolved by the developer during system integration.

The ASCET-SD tool includes target code generation. For Simulink code generation is available in form of the optional *TargetLink* code generator [131]. In contrast to COLA, these tools are not aimed at the generation of distributed code, let alone configuration of the target platform. They are rather limited in producing executable code for non-distributed tasks. Integration of those tasks into the overall system remains a manual step.

4.5.3 Platform Concepts for Distributed Real-Time Systems

In addition to the presented MDD tools, concepts for execution platforms for distributed real-time systems are common. These platforms are intended to ensure safe operation of distributed real-time systems by scheduling the resources available, e. g., processors and busses. This is either achieved by appropriate offline scheduling or enforced by an online monitoring system, similar to the TTA which has already been introduced in Section 2.4.

A concept called *Giotto* for generating target platform schedules for hard real-time systems has been presented by Henzinger in [68]. In Giotto, tasks are modeled by means of their WCET. Operating modes may be defined, activating or deactivating tasks according to the current system state. The models specified with Giotto are also executed in a cyclic manner, similar to COLA models. An extension of Giotto towards distributed platforms has been described in [69], called *Distributed Giotto*. Unlike COLA, Giotto defines the causal order of tasks and their resource requirements, but does not deal with specifying their implementation. Tasks are rather implemented by hand and the Giotto compiler guarantees their timely execution on a specific platform, given worst-case execution times and call frequencies for all tasks are known. An extension for distributed platforms is also available [69]. COLA in contrast defines the tasks implementation, which allows for verifying their implementation and calculating reliable execution times based on the designed model, as presented in [130].

The *BASEMENT* approach [62] also envisions the configuration of the execution platform using a model of software components. These components may be executed either periodic or aperiodic and are called *red* or *blue processes*, accordingly. For the red processes, processor and bus schedules are calculated offline. These static schedules provide deterministic timing behavior at runtime. Resources which remain unused are employed for executing blue processes. For these processes, no timing guarantees can be given. In comparison to COLA, the BASEMENT approach does not enable the functional design of the application software. Its model is meant for configuring the platform only. Consequently, information about the resource requirements of the tasks have to be provided by the developer. In a system based on COLA, this information is calculated automatically from the model. Further, all tasks are scheduled statically which may result in higher processor utilization. On the other hand, the BASEMENT approach proposes the use of separate kernels for red and blue processes, which is not necessary for a COLA system.

4.6 Chapter Summary

In this chapter we introduced the modeling language COLA and the development process based thereon. By providing an integrated modeling concept, which covers all phases of systems development from requirements to functional design and technical architecture, COLA facilitates the design of automotive systems using a single modeling language. Its integrated nature avoids any — either manual or automatic — migration steps between different languages or tools, respectively. This way loss of information and inconsistencies caused by human errors or incompatible modeling dialects can be prevented.

To test the applicability of the COLA concept for automotive systems, a prototypical implementation of a COLA editor has been carried out as well as some plug-ins based on it. We have mentioned the simulator and some of the model checking tools in this chapter. In the following chapter we will introduce the plug-ins for the automatic deployment of a system from COLA.

As we have outlined in the related work section, extensive tool support already exists for MDD of automotive systems. However, we also identified some advantages of the COLA approach compared to others. An approach which cannot be ignored due to its broad support is the AUTOSAR concept. Since more and more ECUs are migrated towards the AUTOSAR standard, any other approach should provide a concept for integration into an AUTOSAR system. We will present such an integration of code generated from COLA in Section 6.3

Chapter 5

Fully Automatic Deployment

In this chapter we focus on the main contributions of this thesis. The results enable the unattended code generation and system integration of automotive applications modeled in COLA for a distributed embedded platform.

First, in Section 5.1 we will introduce those steps of the COLA-based development process, which facilitate the automated deployment of automotive software. Subsequently, the implementation of these steps, namely *code generation*, *allocation and scheduling*, and *system configuration*, will be described in detail in Sections 5.4, 5.5, and 5.7, respectively.

Besides these major steps we want to put some emphasis on several topics being required to make the automatic deployment approach work, but are at first glance hidden in the deployment process. First of all, code generation is dependent on the behavior and interface of the underlying system platform. To this end, we require the platform to consist of a custom combination of hardware, operating system, and a communication middleware. The particular requirements for these components will be given in Section 5.2.

Another essential task during system deployment is the *dependency analysis*, which will be introduced in Section 5.3. The resulting graph structure is used for the assignment of communication addresses and specifies the causal order of tasks needed by the scheduling algorithm.

In Section 5.6 we want to point out the specifics of generating system operating modes. During the actual deployment, these are generated along with all other tasks in the code generation step. To realize an operating mode on the target system, however, also implies a modified scheduling scheme. This is why, we will describe code generation and scheduling for regular tasks, before explaining operating modes.

5.1 Deployment Overview

In Section 4.1 we described our vision of an ideal automotive software development process, based on COLA. Compared to the given description of the entire process, we are focusing on our contributions to the process in this chapter. These contributions enable the deployment of a system modeled in COLA onto a distributed target platform. Figure 5.1 is, compared to the related illustration of the entire process in Section 4.1, limited to those steps relevant for deployment. Again, named arrows in the figure depict the respective actions, being performed automatically by the deployment tools, while resulting artifacts are indicated by the small icons. As can be seen in Figure 5.1, the deployment approach facilitates iterative changes of the modeled system by providing the concept of model-level debugging. This concept is an optional, advanced topic and will be described in Chapter 6.

The result of the presented process is a distributed system which is already integrated and ready to be executed. We will give the rationale for each of the depicted steps in the following:

System modeling: Before deployment can take place, all modeling actions have to be finished. Thus, as an input for the deployment tools, complete Logical and Technical Architectures have to be available. While the Logical Architecture comprehends a model-checked design of the intended system's functionality, the Technical Architecture specifies a partitioning into distributable clusters and provides key figures of the intended hardware platform.

Code generation: As a prerequisite to code generation, the dependencies between the clusters defined in the Technical Architecture have to be known. This knowledge is achieved during a dependency analysis step. The analysis results in a graph structure which indicates all cluster dependencies for the given model. The dependencies are relevant for scheduling all tasks in their causal order. In addition, the graph is used for assigning communication addresses to all channels which realize an exchange of data between different clusters.

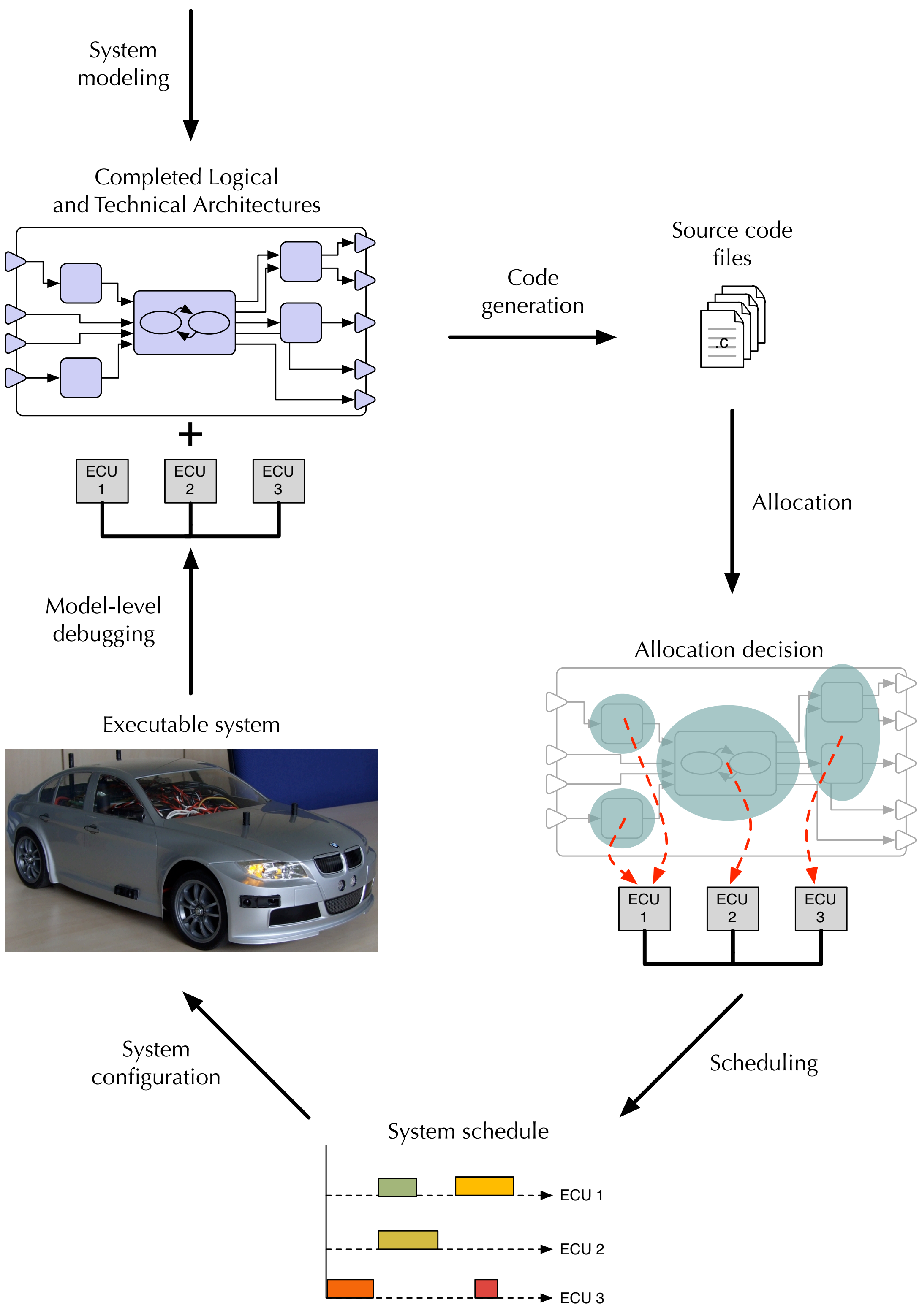


Figure 5.1: Deployment workflow

When communication addresses are known, generation of the functional code can take place. The according code generator transforms each cluster of the system modeled in COLA into a task for the target system. Using only calls to a middleware and its logical addressing for external data exchange, the generated task is independent of its executing target hardware node. The result of the code generation step is a set of source code files which implement the respective clusters. These files may then be compiled to derive executable code for two purposes. First of all, this code is executed in the resulting system. Second, for providing input figures to the allocation algorithm, the code my be employed to determine the task's WCET.

Allocation: Using the concept for performance estimation, which has been presented by Wang et al. [130], the worst-case execution time for each task on each node of the system can be estimated. These figures are used for the definition of a valid mapping between tasks and computing nodes of the hardware platform for their respective execution. Further, the memory requirements as well as other NFRs are considered to derive an allocation scheme. The calculated allocation is stored in the Technical Architecture and is thus available to the scheduling tool and for system configuration.

Scheduling: As soon as the allocation is fixed, the calculation of schedules can take place. Scheduling is done offline according to our deployment concept. The resulting scheduling plan is then executed as defined at system runtime. For this purpose the time-triggered paradigm, as described in Section 2.4 is employed.

If the modeling concept of mode automata, which has been introduced in Section 4.3, is used, the scheduling plan generator constructs a set of schedules for each node of the system. These schedules are then switched at runtime according to the current operating mode of the system. This concept will be explained in detail in Section 5.6.

System configuration: In order to configure the target system, that is, loading the communication matrix into the middleware instances and configuring the schedulers, some configuration files are generated for each node of the system. These files comprehend the defined communication, allocation, and scheduling data for each computing node.

For configuration of the middleware, the addresses stored in the dependency graph structure are matched to the allocation scheme, which is available in the Technical Architecture.

Model-level debugging: Another major benefit of a middleware besides transparent communication is the possible inclusion of additional services. The middleware presented in this thesis defines extensions for the model-level debugging of generated applications. This is facilitated by logging data at runtime and loading them in a model simulator afterwards. Thus, the actual behavior of the target system can be replayed at model level to identify design errors.

Since it is our goal to deploy the model onto a distributed platform, we have some requirements regarding the platform's capabilities. These requirements include the employed operating system as well as the communication mechanism. As for the operating system, first of all the used scheduling concept is of importance. Otherwise it would be impossible, to deploy the system while preserving the model's timing semantics. Further the mode of operation for interrupt handling and hardware access, especially sensors and actuators, have to be known. By specifying a standard for these parameters on all employed ECUs, the functional code can be generated without knowing the physical location of execution on the target platform. Our requirements towards a suitable target platform are presented in the following section.

In summary the deployment concept enables a lossless, automatic transformation of a system modeled in COLA into an executable system. As the COLA model is subject to model checking, these automatic transformation steps guarantee the preservation of system quality, by rendering manual coding redundant. Since the Component Language covers the entire development process, especially language constructs for software architecture as well as hardware architecture, the degree of automation presented here is made possible. The result is a system which needs much less effort regarding system integration. All necessary function calls for basic system services as well as addressing for communication and task synchronization are already ensured by the code generation tools.

5.2 Platform Requirements

The target for our deployment approach is a platform consisting of hardware, operating system, and middleware meeting several requirements. We will give the reasons for these requirements in this section. Figure 5.2 provides an overview of the intended system architecture.

As shown in the figure, the architecture consists of a number of ECUs which are interconnected by a bus system. In addition to the integrated components, like processor, memory, etc., several sensors and actuators may be connected to an ECU. Each of the ECUs comes with an operating system and the appropriate drivers for its respective hardware components. Between the applications executed on each node and the operating system, a middleware layer serves as a mediator, as Figure 5.2 indicates. The middleware provides a common interface to access basic software services like communication and hardware access on every node. By this

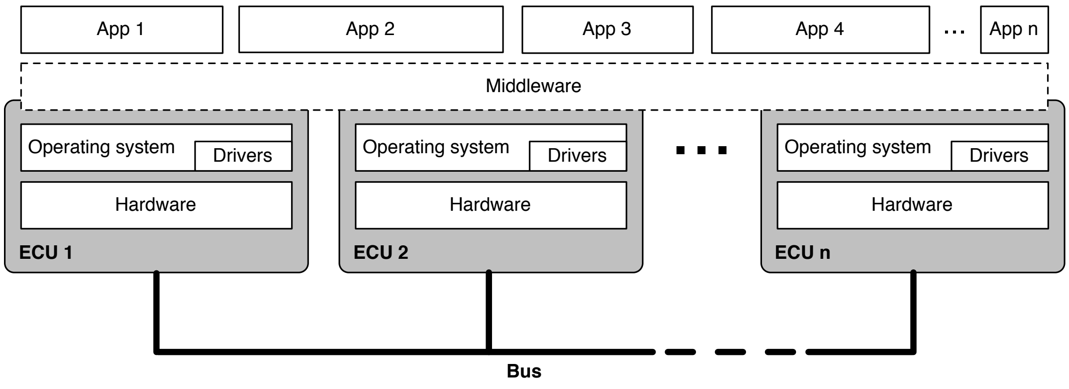


Figure 5.2: System architecture

means it decouples the executed applications from their physical location and, in addition, facilitates the automated generation of code for its fixed API.

We will specify our exact requirements for the target platform, that is, hardware, operating system, and middleware in the following.

5.2.1 Hardware

Regarding the employed hardware, we require the processors and memories, both for program storage and execution, to be of sufficient speed and sizes respectively. Further all other non-functional requirements defined in the COLA model have to be fulfilled by the platform. We will detail the particular requirements during the description of the allocation concept in Section 5.5.1. Our deployment approach checks for these requirements and outputs an error, if the platform is not sufficiently equipped for execution of the modeled system, to prevent a non-operational result. It should be clear that it is impossible for the deployment to find a valid deployment solution, if the platform does not meet the given requirements.

Furthermore, we assume the platform to be suitable for a time-triggered execution of the system. This requires hardware interrupts to be handled by dedicated IO-processors rather than the application processor. Figure 5.3 depicts the required architecture. Each ECU has one or more IO-processors which are connected to the devices associated with the ECU. The IO-processors are responsible for receiving and sending data from and to the sensors and actuators. To this end they have a data buffer for the most recent values — which matches with COLA's last-value semantics — and communicate with the application processor. The IO-processors accept and deliver if and only if they are polled by the processor to do so. This prevents interrupts from influencing the timing of the pre-defined schedule. The same concept applies for the bus controllers used in the ECUs.

Considering communication we require the platform to feature a time-triggered communication mechanism. To achieve time-triggered communication without in-

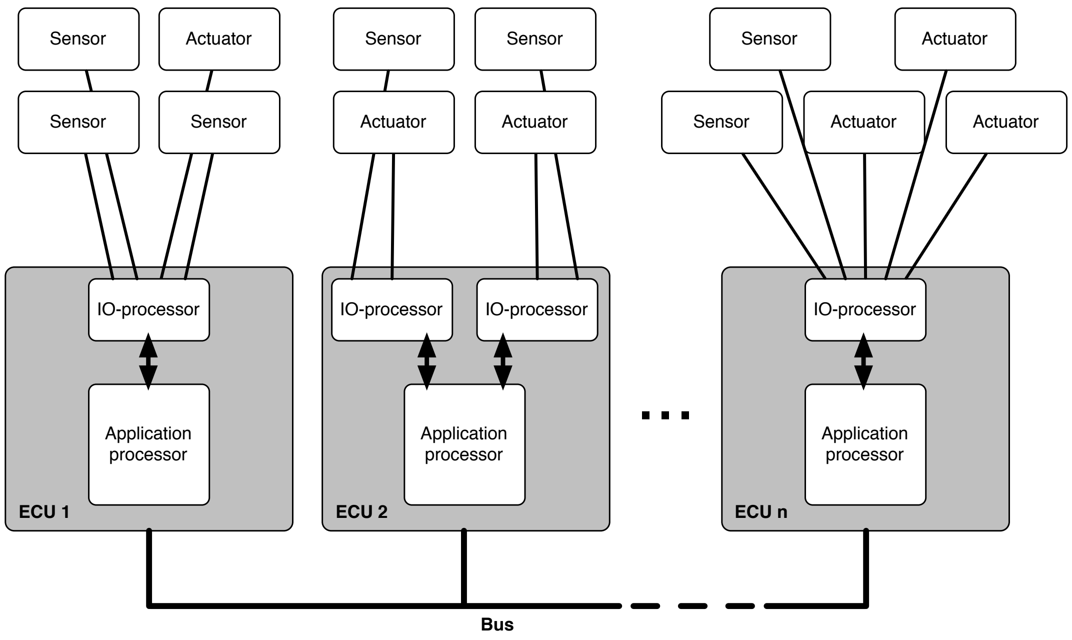


Figure 5.3: System architecture for polling sensors and actuators

terfering with task execution, such a communication mechanism has to be implemented partly in hardware. FlexRay [23] or TTP [82] are prominent examples of buses featuring such a mechanism. For our deployment we require a bus protocol slightly differing from FlexRay and TTP, as we will explain shortly.

Summarizing these properties, the resulting platform is similar to the TTA introduced in Chapter 2.4. Yet, the platform described here differs regarding its bus protocol and middleware, as we will elucidate in the following section.

Bus protocol. In the course of defining a time-triggered system, we require the bus system to provide time-triggered communication. Current bus systems like FlexRay [23] or the TTP [82] implement the time-triggered paradigm by means of a TDMA protocol. That means, access to the network is divided into time slots. Within each time slot only one node is allowed to send data, all others may receive data during that time. All slots together define the TDMA cycle. Figure 5.4 shows such a cycle, divided into five slots. Each node can be assigned multiple slots depending on the communication needs of the tasks it executes. An example for this is shown in Figure 5.4 where two slots are reserved for node 0. The TDMA cycle is repeated periodically and its duration should be chosen long enough to facilitate all nodes to finish their respective tasks. This allows for the definition of a system wide scheduling cycle. Tasks which have to be executed at a higher frequency can be involved several times during each cycle.

Two approaches for allocating a time slot to a sender can be thought of. One

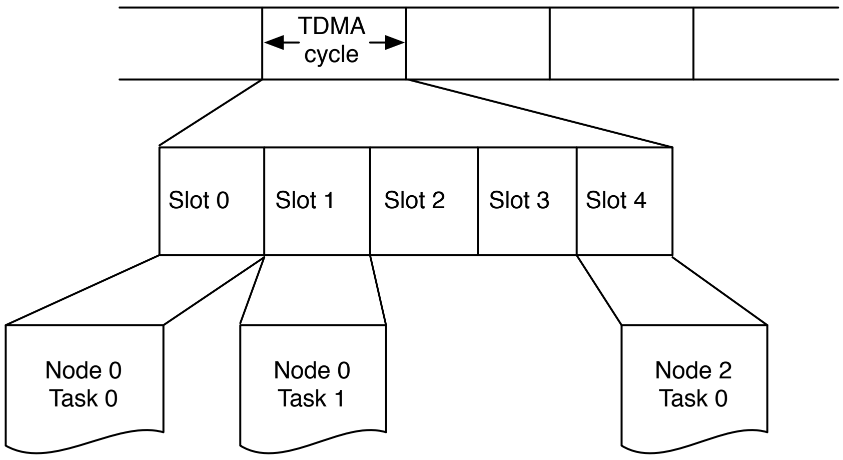


Figure 5.4: TDMA scheme

possibility is that each node in the system is assigned one time slot in each TDMA cycle, in which the node is allowed to send data. The length of that time slot either depends on how much data has to be sent by that node or the available amount of time is simply divided by the number of communication nodes.

Another approach of distributing slots would be to assign a time slot to every task. This leads of course to more slots in a cycle and a longer cycle time, but the delay between finishing a task execution and distributing the produced data can be shortened. Therefore task scheduling and TDMA time slots have to be aligned to minimize distribution delay, shown in Figure 5.5. For our middleware the latter option of assigning slots to tasks rather than to nodes is chosen due to it's greater flexibility in generating schedules and advantages considering delay. This is why, we require the protocol to provide a suitable configuration mechanism.

Besides that, our deployment enables the use of different system operating modes which typically imply a change of the active tasks, and hence the communication schedule. To this end the bus protocol has to distribute information about the actual operating mode and facilitate a method for reassigning TDMA slots to different senders at runtime. TTP features support for operating modes in the protocol itself, but limits the maximum number of modes to eight, which is not sufficient for our approach. As for the reassignment of communication slots, TTP is not designed to enable an arbitrary number of changes of communication schedules, but allows one schedule for each operating mode.

The COLA deployment does not specify an upper limit for the number of operating modes. Thus a suitable communication protocol has to provide the basic concepts of TTP like time-triggered operation, fault containment regions, and clock synchronization, but at the same time should not limit the number of different TDMA schedules. The deployed system is then able to communicate operating

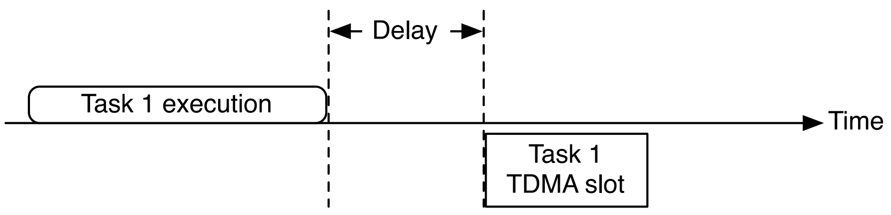


Figure 5.5: Delay between task termination and data distribution

modes just like regular application data and reconfigure the bus system in case of a change of the operating mode. For a more detailed introduction into the deployment of operating modes refer to Section 5.6.

5.2.2 Operating System

In order to ensure the preservation of the COLA model's time semantics during execution on an actual target platform, we require the operating system to conform to our requirements for interfacing hardware and scheduling tasks. Regarding the hardware requirements, we demand a fixed API for all drivers as we will outline in the next section. Afterwards, we give our specification of a suitable task scheduling scheme for the execution of tasks generated from COLA.

Driver concept. Today it is a well-established procedure to use custom interfaces for the drivers in an automotive system. Since systems were manually coded from scratch, this concept did not influence system quality in a negative way. However, reuse of existing application code was limited as it had to be adapted to each hardware platform and its particular driver interfaces. Similarly, changing the employed driver later on, for example if the hardware was replaced by an appropriate alternative part, was impossible without changing the related application.

Due to the increased use of embedded computers across automotive product lines and the growing number of companies offering ECUs and sensors/actuators, the reuse of code and hardware is seen as a key factor for lowering development cost nowadays. In order to enable the reusability of application code, a defined standard interface between the application and lower system layers, that is, operating and communication system as well as hardware drivers is needed. Such a standardization facilitates interchangeability of software and hardware elements. At the same time a mixture of hardware suppliers can be entrusted with the delivery of alternative parts. If their operation as well as their driver interfaces are equivalent, the resulting cars perform identically. This enables the OEM to profit from the arising competition between the suppliers.

The AUTOSAR initiative chose an approach which can provide the necessary standardization of interfaces. The employed RTE acts as a layer between applications and operating system, or drivers, respectively. But the RTE does not rely on a fixed API. Instead it is rather a modeling concept which is transformed into code by the RTE generator. If used consistently, the API will look the same to all applications, no matter which hardware is used. However, if the RTE is modeled separately for each system, interfaces are not necessarily compatible. In this respect the RTE code generated for an AUTOSAR model is rather an adapter than a middleware.

For our deployment we decided to use a middleware with a fixed API for two reasons. First, this API provides a fixed interface for the COLA code generator, which eases implementation of the code generator itself. Second, a change to a system generated from COLA does not require the middleware to be replaced, if the interface is modified. Rather, a new configuration may be loaded into the middleware to enable the integration of a different task set. In addition to these properties, the implementation of the middleware should be usable in combination with a lot of different platforms, that is without requiring changes in the source code.

The configuration of the middleware according to a specific platform is achieved by a configuration file which is read by the middleware at system start-up. Besides bus addresses, the configuration file also specifies the hardware devices to use. For interfacing these devices the middleware requires a UNIX-like driver interface, that is, the drivers can be opened like a file and accept or provide character input or output. Stream-based data exchange is not envisioned at the moment since most sensors and actuators use only small datagrams of fixed size for communication. In principle, of course, the middleware could be extended towards stream-based communication. The assumption of small datagrams is valid since sensors are becoming more and more intelligent subsystems, which are able to provide an A/D conversion of input data, filter glitches, and even fusion data of several sensors before delivering their results to the ECUs. This evolution and possible architectures thereof have been presented, amongst many others, by Luo and Kay [93], and Meier [101].

As the deployment concept aims at creating time-triggered systems, sensor drivers shall not generate interrupts. Rather, the middleware polls data whenever necessary. This polling is achieved by issuing a read command to the according driver. When this command is received, the driver shall return the most recent values.

Task scheduler. For the task scheduling of the underlying operating system we require time-triggered, non-preemptive scheduling. This assumption is made by the scheduling tool described later in this chapter to calculate the overall system schedules. During execution, the nodes' schedulers serve only as dispatchers, that is, the schedulers do not decide on which task to execute, but rather read the tasks and their starting times from a schedule which is generated offline by the scheduling tool and copied onto the ECUs. The operating system scheduler is just liable to release the tasks at the calculated points in time.

The rationale for employing a time-triggered approach is its natural fit with synchronous languages. Benveniste et al. characterize the TTA as a synchrony compliant architecture in [19], which also employs time-triggered scheduling. Just like a synchronous model which is based on the cyclic execution of software components, the TTA is based on a fixed schedule which is executed over and over again. Thus, it is able to meet the timing requirements demanded by a synchronous model. In addition, the time-triggered approach is favored instead of event-triggered concepts for hard real-time systems. Amos Albert gives a good overview about the advantages for distributed system of both approaches in [3]. While event-triggered scheduling leaves some uncertainty regarding system behavior at runtime, time-triggered scheduling can be guaranteed to show deterministic time response.

5.2.3 Execution Middleware

Several middleware implementations for distributed embedded systems have been proposed over the years. The reason for designing a custom middleware for the deployment approach presented here, lies in a lack of flexibility of the existing concepts. Considering for example the current state-of-the-art, namely the AUTOSAR RTE, its addressing is fixed at system creation. By generating hardcoded communication channels from the AUTOSAR model, a subsequent change regarding the placement of tasks onto ECUs or the design of their interfaces is impossible. Thus, it might be necessary to re-program more than a single ECU in case of a change. On the other hand, the middleware presented here is configured at system start-up and might therefore be reconfigured simply by copying the new parameters to the ECUs and restart the system. Besides providing a convenient way of bug fixing, this flexibility will aid in changing application software components during system lifetime, which is one of the future goals for automotive systems, as presented for example by Pretschner et al. [108], and Heinisch and Simons [67].

An advantage of the RTE is that it already defines a lot of functions for system management and diagnostics. These functions are a necessary part of automotive systems. The middleware presented here does not yet feature such an extensive API. Instead we focus on the transmission of messages between tasks and hardware, as well as task data storage in the system, which are the essential facilities for our deployment. For productive use, the addition of management and diagnostics functionality would however be a necessary extension.

We will give a short introduction into the functionality of the middleware, which we implemented for the COLA deployment, in this section. The middleware has first been described in [53].

Local and remote communication. A basic function of each middleware is the provision of communication mechanisms. For a clustered COLA model this includes inter-cluster communication. The border lines separating clusters are crossed by communication channels. Each such channel indicates a need for data exchange at runtime. This is one of the duties of our middleware. Every channel is assigned a virtual address, which is inserted into the appropriate middleware read and write calls during code generation. The middleware distinguishes between local and remote communication, according to the placement of sender and receiver. During execution remote communication is then accomplished using the underlying bus system, while local communication is achieved by buffering messages in the local middleware instance.

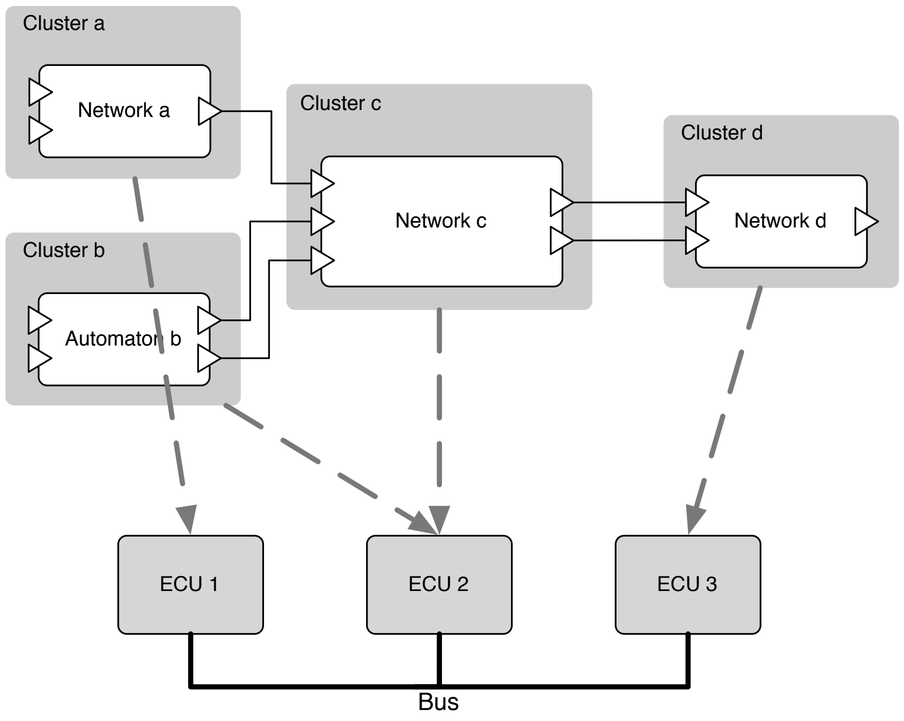


Figure 5.6: A possible allocation of clusters to ECUs of the target platform

Figure 5.6 shows an example allocation of clusters to ECUs of the target platform. As indicated by the gray arrows, `Cluster a` is allocated to `ECU 1`, clusters `b` and `c` are mapped to `ECU 2`, and `Cluster d` is mapped to `ECU 3`. That is why the channels connecting the clusters have to be replaced by middleware communication during execution of the system. Since `Cluster b` and `Cluster c` are executed on the same ECU, their data are communicated locally by the middleware using some buffers. The channels connecting the remaining clusters are mapped to bus communication, since the respective clusters are run on different ECUs.

Synchronous modeling languages assume tasks to be executed periodically. In COLA tasks containing either a delay or an automaton are stateful, as described in Section 4.3.3. Besides enabling communication between tasks, the middleware also provides data keeping for all stateful tasks. The necessary middleware API calls are inserted into the functional code during code generation.

Hardware interaction. In COLA several channels may read data from the output port of a unit. If this unit is a source this indicates the value of a sensor to be used by two different software components. Following the synchronous semantics of COLA, both software components have to be provided with the same value. Additionally the hypothesis of perfect synchrony assumes the complete system to be executed in virtually no time. If several sources are read or actuators are written, semantics claim this to happen instantly. This requirement could not be fulfilled if hardware would be accessed directly by each task, as tasks are executed sequentially on each node of the system. This might lead to a significant advance in time between the data access of two consecutive tasks. The same is true for one task which reads the inputs of several sensors, as described in Section 4.3.5.

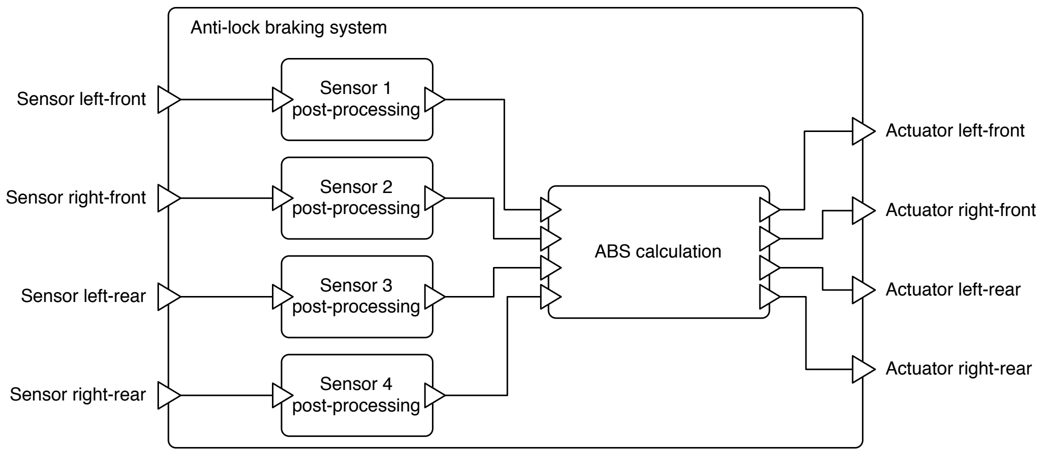


Figure 5.7: ABS example

Considering for example the implementation of an anti-lock braking system in Figure 5.7, the example unit gathers input from the speed sensors of all four wheels to check if one of the wheels is blocking. If that case occurred, the system would adjust the braking power at the respective wheel accordingly. In our example the system is implemented by five units, one for each input from the wheels, and the fifth unit for comparing the values and calculating an adjustment of the braking power, if necessary. It should be evident that for a correct function of the system, the data of the wheel sensors have to be acquired at the same point in time. In the same way, the actuators have to be modified in parallel to achieve the desired effect. If, however, the five units were contained in different clusters, they might be distributed over different ECUs of the target platform, for example one ECU for the front sensors and one for the back sensors, in an arbitrary manner. To retain the model assumption of a zero time data acquisition, which means the data would be acquired at the same moment, a special polling scheme has to be employed by the middleware. This scheme starts with the acquisition of sensor data during each execution cycle, following the clusters are executed, and at the end of the cycle all actuators are written. This scheme guarantees that corresponding sensor data are recorded at the same time.

Figure 5.8 shows this invocation of sensors, tasks and actuators during a scheduling cycle. As the middleware uses last-value semantics, sensor values can be accessed several times by different tasks. Each task will then read the same value.

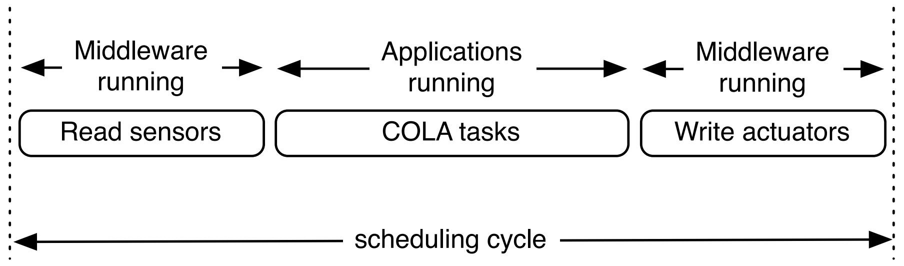


Figure 5.8: Hardware interaction

Global time. The realization of a time-triggered system relies on the availability of a global time, as described in Section 2.4. According to this global time, each node can determine the start of the communication slot(s) assigned to it, as well as the points in time data sent by other nodes in the system have to be received. Further, the global time is used to coordinate the execution of tasks on the different nodes according to the globally defined schedule.

The middleware provides access to the global time maintained by the bus protocol, featuring the `mw_global_time()` call. This function can be used by the nodes' local dispatchers to initiate execution of the tasks. The synchronization of the global time has to be provided by the underlying bus protocol. The middleware simply provides access to this time base for the executed applications.

Middleware API. For transmitting data the middleware does not rely on bus addresses of sender or receiver nodes. Instead every datum is assigned a unique numerical identifier used as its logical address. When a datum has to be transmitted to other nodes, it is passed to the middleware in combination with its logical address. Using this identifier, the middleware can determine which TDMA time slot has to be used for sending. As the TDMA cycle reaches the determined time slot, the datum with its identifier is broadcasted over the network. The receiving nodes store the received datum with the identifier as its key. Already existing data with the same key are replaced with the received data. Tasks on the receiving nodes are now able to retrieve the datum from the middleware by passing the identifier.

The configuration for each node's middleware instance is achieved by reading a configuration file during start-up.

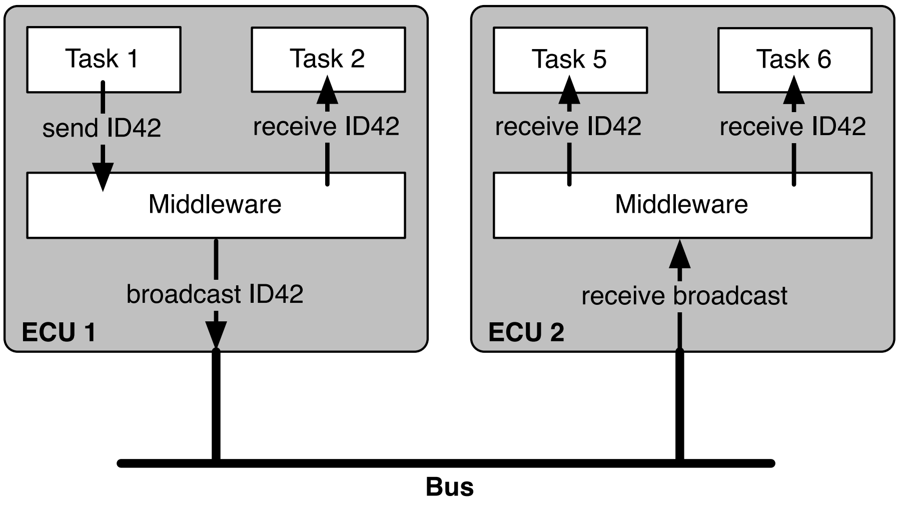


Figure 5.9: Send and receive data using the middleware

The scheme of two nodes communicating is shown in Figure 5.9. `Task 1` on `ECU 1` sends data with identifier `ID42`. The data is stored in the middleware, `Task 2` on the same node has immediate access. After the broadcast (which might be some time later), the tasks on `ECU 2` also have access to the data sent by `Task 1`.

```
mw_send(&variable_y, sizeof(variable_y), 17);
mw_receive(&variable_x, sizeof(variable_x), 16);
```

Listing 5.1: Inter-cluster communication

To achieve a small and simple API, the middleware only provides the basic functions `mw_send()` and `mw_receive()` for inter-cluster communication. The first argument is a pointer to a variable the data are read from or written to, while the second argument specifies the amount of data to read or write. A logical address identifying which data to send or receive is passed to the function as third argument. An example for the application of these calls is shown in Listing 5.1. The listing depicts the code for some fictive variables named `variable_x` and `variable_y` which are read and written, respectively.

```
mw_restore_task_state(&unit_state, sizeof(unit_state), 7);
mw_save_task_state(&unit_state, sizeof(unit_state), 7);
```

Listing 5.2: Task state storage

For restoring and saving a task's state the calls `mw_restore_task_state()` and `mw_save_task_state()` are used. The usage of these calls is shown in Listing 5.2. The functions' arguments are identical to those of `mw_send()` and `mw_receive()`.

The middleware's functions for inter-task communication may also be used to exchange data between a task and a device, as we will describe next.

Interfacing sensors and actuators. Since a real-time computer system has to interact with the real world, there is a need to connect the tasks with sensor and actuator devices. These devices can be attached to any node in the system. The middleware provides functionality for transparent handling of remote devices to all nodes. Therefore sensor data are presented to the task like all other data, as described in the previous section. That means every sensor is assigned a unique identifier, the middleware is in charge of polling the sensor in an appropriate cycle time and storing the result with the identifier as a key. To make the data available for remote nodes the distribution mechanism for data is used.

Writing data to actuator devices works similar. Data are stored in the middleware and distributed. When the node with the actuator device attached has to write a new value, it looks up the data for the identifier associated with the actuator device and writes it to the device. Sensor and actuator interaction is completely handled by the middleware, tasks just have to use the middleware's `mw_send()` and `mw_receive()` API functions to write and read values.

As hardware device interaction is handled by the middleware and not by tasks directly, a scheme is needed defining when this interaction takes place. The scheme is indicated in Figure 5.8. At the beginning of a cycle the middleware first reads sensor devices. Each sensor is read according to a given frequency. This means a sensor could be read every n-th cycle, in all other cycles the last value read will be used. After sensor values have been read and distributed, they are available to all tasks on all nodes. At the end of the cycle, when all tasks have finished execution, each node with an actuator connected checks for data with the ID of the actuator. If such data are found in a buffer, they are written to the hardware. The update frequency of actuator values can again be specified in the COLA model. This way the middleware does not try to write the actuator more often, avoiding race conditions in the actuator's data buffer.

Since the middleware instances on all nodes of the system are synchronized, the first scheduling cycle should start with a synchronous sensor reading of all middleware instances. Thus, each cluster is provided with its respective input values. Subsequently, for each actuator there is a calculated value at the end of the scheduling cycle. This scheme would render initial values for sensor and actuator data unnecessary. However, to guarantee a safe operation of the system in case one sensor could not be read or an ECU does not provide a value to an actuator due to a defect, all sensor and actuator buffers are initialized with default values.

These values are taken from the COLA model and encoded in the middleware configuration.

Middleware configuration. The middleware can be configured using a file. It contains an element hierarchy of nodes, tasks and data. Data identifiers can be defined and assigned to the tasks they belong to, tasks are sub-elements of nodes. TDMA slots can be assigned at task level, i.e., all data belonging to one task will be sent in the slot assigned to that task. Further, sensors and actuators are assigned to nodes so that the middleware can perform hardware access on the proper node. Sensor and actuator elements are like data elements with some hardware specific information, i.e., they also have identifiers for send/receive operations. Additionally a TDMA slot has to be assigned to hardware devices for sending values, because they operate independent of tasks as shown in Figure 5.8.

With this configuration information, the middleware is able to allocate buffer space during the startup phase, initialize sensor and actuator hardware for operation and distinguish between local and remote communication. The exact syntax of the middleware configuration file will be explained in Section 5.7, which deals with the configuration of the target platform.

5.3 Dependency Analysis

An elementary step in our system deployment concept is the analysis of cluster dependencies. These dependencies provide the basis for logical address generation, communication buffer allocation, and the calculation of valid system schedules. In order to capture the dependencies in an easily processible way, we created the *Cluster Dependency Graph* (CDG), which has been first presented in [84].

Our graph construction algorithm parses the Technical Architecture of a COLA model to derive the appropriate CDG. The resulting graph consists of vertices representing either clusters or communication buffers, and edges indicating data exchange between the former. The CDG is a directed and acyclic graph, thus illustrating the direction of the dataflow. We will introduce the different graph elements in the following, using the graphical representation of the example graph in Figure 5.10. The graph is taken from one of our case studies, which will be presented in Section 7.1.

Working cluster vertices: For every working cluster of the model an according vertex is created in the CDG. The graphical representation of a working cluster vertex is a rectangle. These clusters receive and send messages from and to other clusters. Each of these messages, which are exchanged along channels in a COLA model, are routed along the edges of the CDG. `Radar`

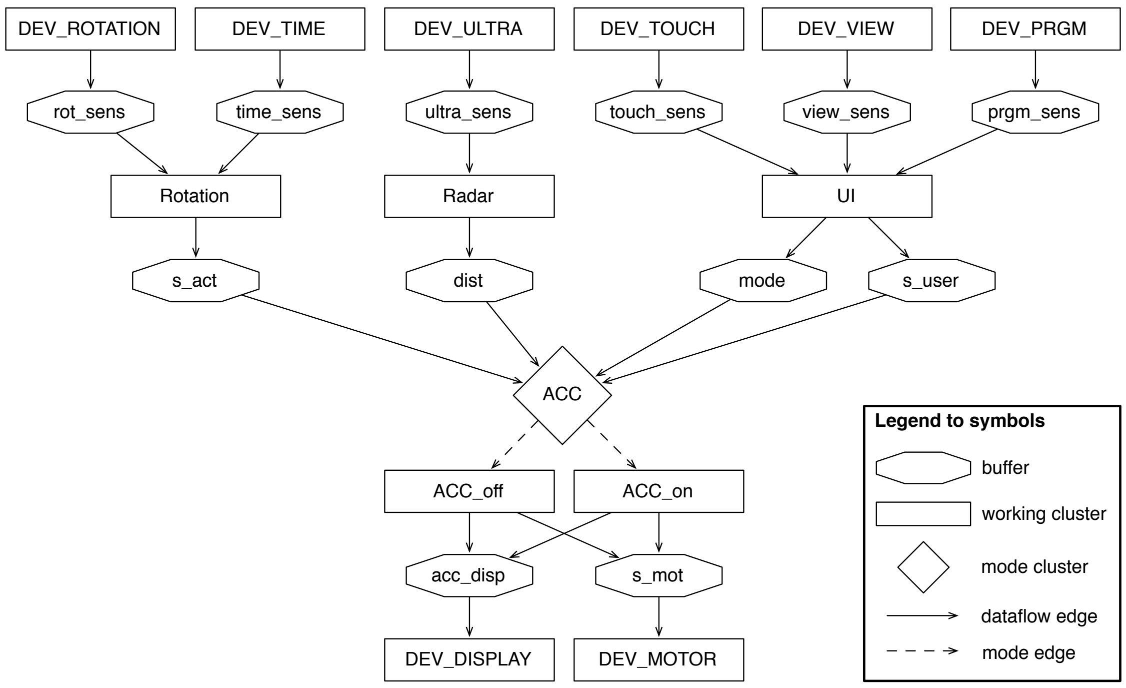


Figure 5.10: A cluster dependency graph

and `Rotation` are two examples for working cluster vertices in the given example.

Some of the shown working cluster vertices carry the prefix `DEV_` in their names. This prefix indicates the clusters to model a device, that is, a source or a sink. Devices are represented as working clusters, too, since they emit and absorb values which have to be transmitted by our middleware. It is therefore necessary to give these devices an address, and provide a buffer for the respective values. Hence their inclusion in the CDG.

Mode cluster vertices: Mode clusters vertices stand for the mode clusters of the model. Each mode cluster indicates a possible change in the control flow of the model, thus deciding on which working clusters to executed subsequently. This determination is pictured in the CDG by subordinating the mode cluster vertex the cluster vertices it activates or deactivates. These subordinate clusters may be either working or mode clusters. The diamond shaped `ACC` is an example for a mode cluster vertex in Figure 5.10. It determines whether `ACC_on` or `ACC_off` has to be executed.

Just like a working cluster vertex, the mode cluster vertex has ingoing edges which represent received messages. The cluster uses these data to calculate which subordinate cluster to execute. In contrast to working cluster vertices however, the mode cluster vertex does not emit an arbitrary number of messages, but always exactly one. This message represents the result of its mode decision.

The input data available to the subordinate clusters are identical to those received by the mode cluster.

Buffer vertices: Whenever data are communicated between clusters, these data have to be stored temporarily by the middleware. Since one of the goals of the CDG is the generation of logical addresses for the middleware, the according data buffers for temporary storage of messages exchanged between clusters, are also present in the CDG. To this end each connection between two working cluster vertices contains a buffer vertex.

In Figure 5.10 the buffer vertices are depicted as octagons. Typically, buffer vertices have one incoming and one outgoing edge. If, however, a buffers stores data which are read by several clusters, the buffer vertex may have more than one outgoing edge. It also may feature several incoming edges, if more than one cluster writes the buffer. However, this is only possible if the writing clusters are in different operating modes and thus can't be active at the same time. `ACC_on` and `ACC_off` are an example for such alternatively activated working clusters. Depending on the result of mode cluster `ACC`, either `ACC_on` or `ACC_off` is executed and, hence, only one of them writes its results to the buffers `acc_disp` and `s_mot`. So despite the possibly multiple incoming edges there are no race conditions at a buffer vertex.

Dataflow edges: The CDG's vertices are either connected by dataflow edges or mode edges. Dataflow edges do represent data exchange from clusters to buffers or the other way round. Their direction indicates whether a buffer vertex receives or emits data via the edge. If a vertex has more than one incoming or outgoing dataflow edge, each edge represents a connecting channel between the according cluster and its environment. Clusters which only have outgoing edges contain sources, while clusters containing sinks only have incoming edges.

Examples for such clusters can be seen at the top and bottom of Figure 5.10, respectively. The according dataflow edges are depicted as solid arrows in the figure. Their direction matches the direction of the dataflow.

Mode edges: In opposition to dataflow edges, mode edges represent a means of control flow. To this end, a set of mode edges is directed from each mode cluster vertex to its subordinate cluster vertices. Each of the those subordinate cluster vertices together with its descendent cluster vertices forms an operating mode of the system. Thus the mode edges stand for an exclusive activation of the connected working clusters.

The working cluster vertices `ACC_on` and `ACC_off` are examples of such working clusters in Figure 5.10. The according mode edges are depicted as dashed arrows. They indicate the decision for the current operating mode made by the mode cluster `ACC`.

To improve the graphical cleanness of the graph, dataflow edges are omitted for working cluster which have ingoing mode edges. `ACC_on` and `ACC_off` are an example for this: as they both have the same input data as their superordinate mode cluster `ACC`, the according dataflow edges are left out in Figure 5.10.

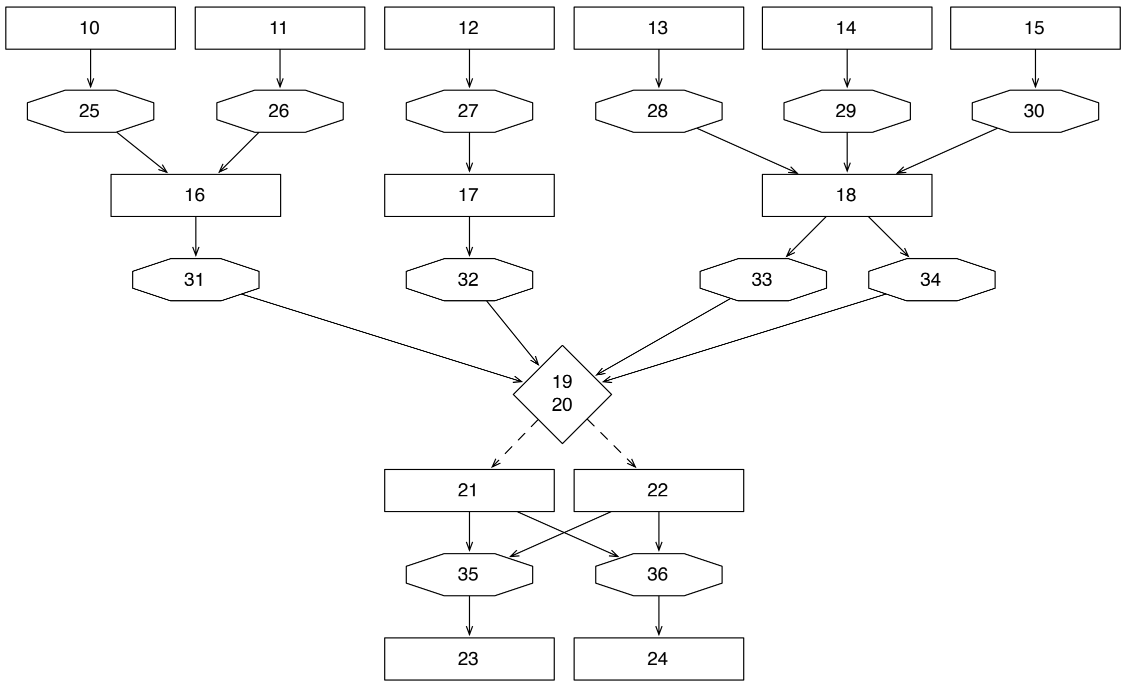


Figure 5.11: A possible address generation result

When the CDG construction is finished, the assignment of logical addresses for communication via our middleware can take place, based on the graph structure. For each working cluster vertex a distinct numerical address is assigned. This address is used for storing the internal state of the associated cluster during system execution. Further, each buffer is assigned an address which the code generator uses to write and read data that are exchanged between clusters. As can be seen in Figure 5.10, mode edges do not contain any buffer vertices. Since the only output of a mode cluster is its decision on the current operating mode, there is no need for an arbitrary number of output values. Rather than inserting buffer vertices into all mode edges, the mode cluster vertex itself is assigned two addresses. One for its internal state and one for its mode decision result. A possible result of the address generation process for our example graph is shown in Figure 5.11, stating the according addresses for all vertices of the graph.

The algorithm for calculating time-triggered system schedules, also uses the dependency graph for analyzing the causal order of clusters to be executed. Basically it follows all the paths in the graph from top, i. e. the sensors, to bottom, which depicts the actuators, and derives a suitable execution order of the according tasks.

5.4 Application Code Generation

The input for the application code generator is a self-contained COLA model. That means that all networks and automata are refined all the way down to the block-level and the model is partitioned into clusters. It is the task of our application code generator to consider the clusters one by one and generate C code for each of them. The result is a set of source code files which are, after compilation, ready for execution by the system as tasks during runtime. For that reason we will use the terms cluster and task interchangeably, here. The following explanation is considering the generation of code for working clusters. We will describe the code generation for mode clusters in Section 5.6 and explain how they modify the active schedule to alter the set of working clusters invoked.

The top-level unit of each working cluster will most certainly be an automaton or a network, since clustering single blocks is not advisable due to the enormous overhead implied by scheduling a plethora of such simple, atomic operations. In the majority of cases this automaton or network will have several input and output ports, which implies some inter-cluster communication.

The code generator will start its traversal of the working cluster beginning with the input ports of this top-level unit and follow the respective connected channels. As soon as it hits another network or automaton, it will descend into the model hierarchy and produce code for the implementation of this unit before evaluating the other units at the same level. This recursive operation resembles to a depth-first search as used for algorithms based on graphs. The light gray track in Figure 5.12 indicates this search path for a COLA example. In the example the algorithm first hits `Unit 1.3` and starts descending to its sub-units. A resulting sequence of encoded units could be:

`1.3` → `1.3.1` → `1.3.1.1` → `1.3.1.3` → `1.3.1.2` → `...` → `1.3.2` →
→ `1.3.2.1` → `1.3.2.2` → `...` → `1.4` → `...`

Subsequently, `Unit 1.1` and `Unit 1.2` would be evaluated, accordingly.

Code generation for the cluster is finished, when the search algorithm has reached every output port of the cluster, providing subsequent clusters or actuators with input data.

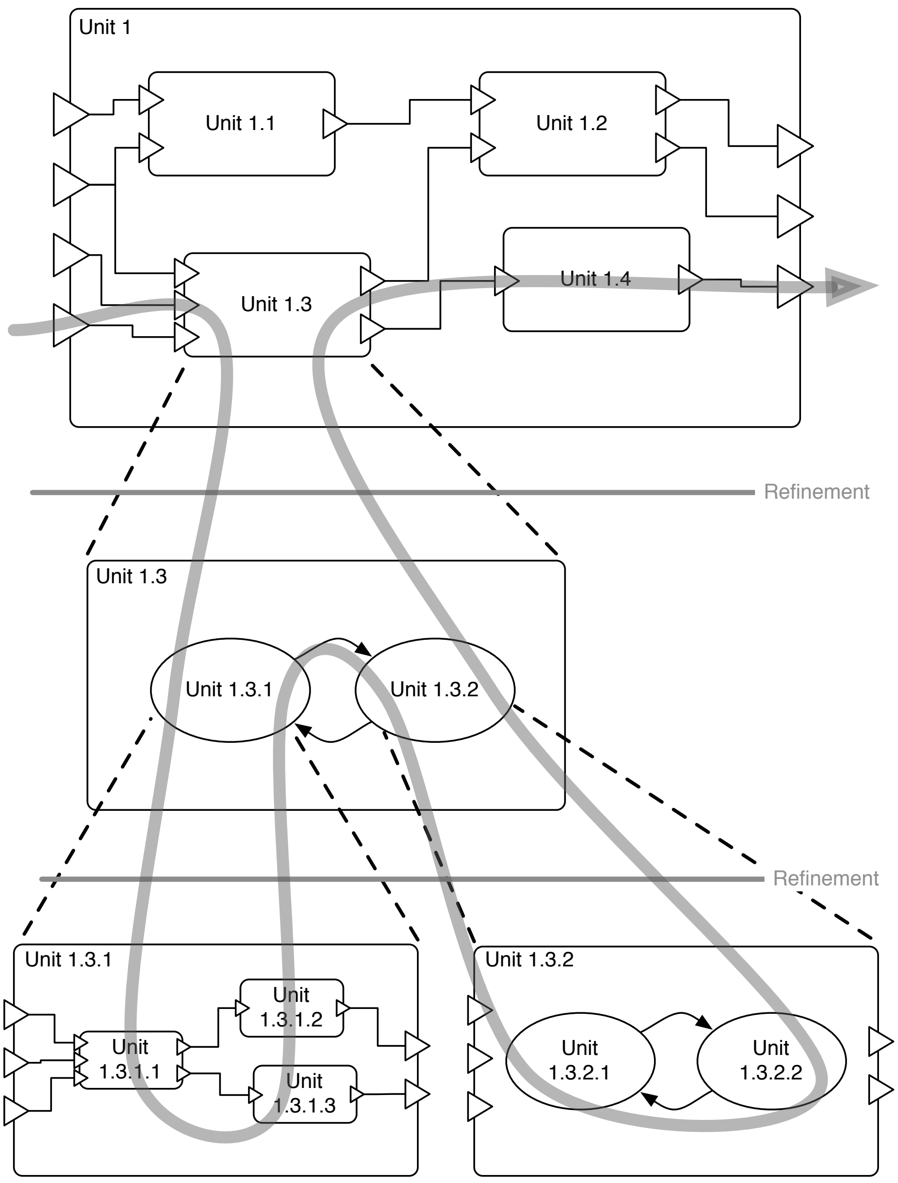


Figure 5.12: Unit evaluation order

5.4.1 Inter-Cluster Communication

As mentioned before, all channels extern to a cluster, that is, those which connect the cluster with either a source, a sink, or with another cluster, imply a need for communication. This communication is realized by our middleware. The code generator inserts appropriate calls to the middleware's API during code generation. Likewise the code generator inserts calls for restoring and saving the task's most recent state. This state is updated during each execution of the task.

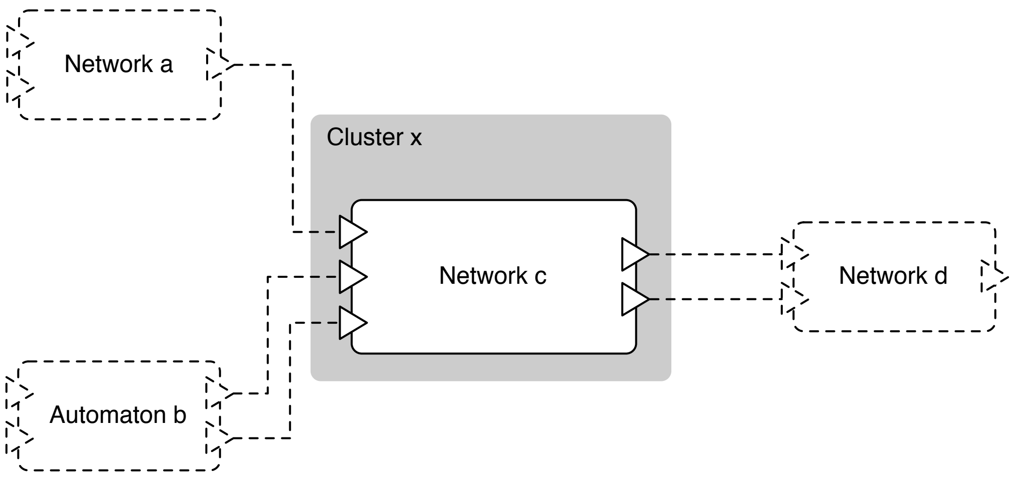


Figure 5.13: Example cluster x

Figure 5.13 shows a snippet taken from a COLA system. In this example `Cluster x` consists of `Network c`, which features three input and two output ports. The channels connecting the network with `Network a` and `Network d` and `Automaton b` are mapped to middleware communication by the code generator. A shortened example of the resulting code is given in Listing 5.3.

```
 1 void cluster_x() {
 2
 3    ... <variable declarations> ...
 4
 5    mw_restore_task_state(&unit_state, sizeof(unit_state), 7);
 6    mw_receive(&var_a, sizeof(var_a), 16);
 7    mw_receive(&var_b, sizeof(var_b), 19);
 8    mw_receive(&var_c, sizeof(var_c), 29);
 9
10    ... <cluster implementation> ...
11
12    mw_send(&var_x, sizeof(var_x), 42);
13    mw_send(&var_y, sizeof(var_y), 66);
14    mw_save_task_state(&unit_state, sizeof(unit_state), 7);
15 }
```

Listing 5.3: Inter-cluster communication

As can be seen in the example, the code generator creates a method named after the cluster. The first step inside the method is a call to the middleware, namely `mw_restore_task_state()` shown in line 5, which returns the most recent internal state of the cluster. Subsequently the input ports of the cluster are read by calling the `mw_receive()` function of the middleware. As explained in Chapter 5.2.3, both API functions take a numerical address and a pointer to the intended variables as arguments. After reading all inputs, the acquired values are used by the cluster's implementation to calculate results. For the cluster's state as well as all input and output values, local variables are declared inside the method for storing the respective values during execution.

After the implementation is fully evaluated, the results are written to the output ports of the cluster. In order to enable the middleware to forward these values to another cluster or an actuator, the respective values are copied to the middleware using `mw_send()` calls. An example of these calls is shown in lines 12 and 13 of our example listing. The last operation of the cluster is the storage of its modified internal state to the according middleware buffer. This is achieved by the `mw_save_task_state()` call exemplified in line 14.

The calls for reading input and writing output values are identical, no matter whether the communication partner is local or remote, a hardware device, or another cluster. The middleware takes care of either buffering the values locally or relying them over the bus system or to a connected sensor or actuator. The generated cluster code is that way independent of the location of its execution. This portability is also important for the estimation of a cluster's worst-case execution time, which has been realized in the SciSim tool presented by Wang et al. in [130]. It allows the clusters' execution on any target system, either real or — in case of SciSim's SystemC platform simulation — virtual, as long as the target provides the common middleware API.

It should be pointed out again, that the middleware uses last-value semantics. This means that reading a value is non-destructive and thus, if a cluster is executed at a higher call rate than the preceding cluster which delivers its inputs, the cluster is still enabled to read valid data. Together with the initialization of all the middleware's buffers for sensor values and the causal order of clusters guaranteed by the time-triggered schedule, there is never the risk of reading uninitialized input values. For the same reason the source code file generated for a cluster always contains a method which is responsible for initializing all stateful units contained in the cluster. This applies to all automata and delays nested in the cluster. The values for this initialization are taken from the COLA model and account for the cluster's initial internal state.

5.4.2 Networks

Cloning the hierarchical structure of the COLA model, the code generator creates a separate function for every network of the respective cluster. As described before, the top-level unit of a cluster is translated into a method without parameters. All input and output values are communicated using middleware calls.

In contrast, all networks and automata on lower hierarchy levels are coded as functions featuring in their signature one parameter for each input or output port. These parameters are realized as pointers for two reasons. First, nesting units according to the model hierarchy would lead to a good amount of unnecessary memory consumption, if the parameters were passed by-value, rather than by-reference. Second, for the output ports it is a convenient possibility to realize several return values, instead of the single value allowed for functions in the C language.

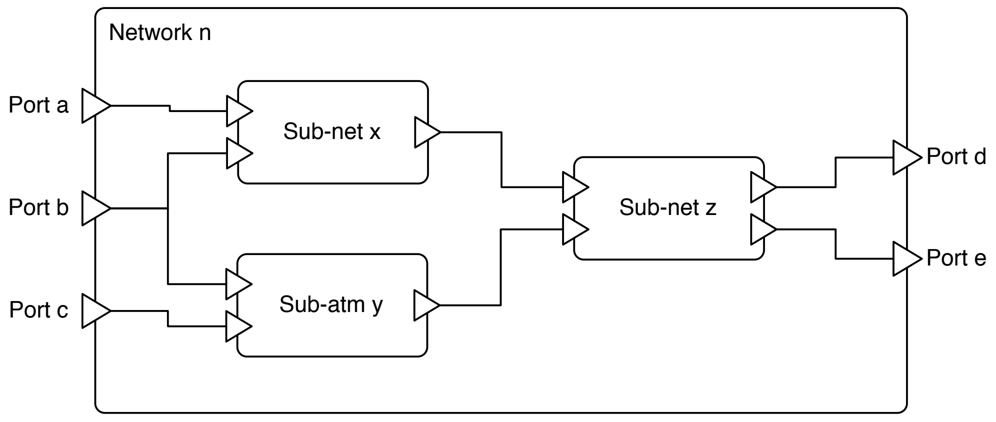


Figure 5.14: An example network

Listing 5.4 provides an example of the code generated for the network shown in Figure 5.14. As can be seen in the Listing, all ports are cited in the functions parameter list. Further, a struct containing the network's and its sub-units' states is added to the parameter list. This struct holds the values of all automata and delay blocks, which are nested inside the actual network, and thus make up its internal state. For all delay blocks, the struct exhibits primitive values, while for all sub-networks or sub-automata the struct features nested structs with the respective values. If, however, the network only contains stateless sub-units, that is no automata or delays, the network itself is stateless and the struct is omitted. A definition of stateful and stateless units has been given in Section 4.3.3.

For executing the sub-units defined inside a network, the evaluation order induced by their connecting channels is important. Of course, the sequence of calls has to preserve the order induced by the semantics of the dataflow, that is, for each unit to be executed its inputs have to be available. To guarantee this, the set of sub-units provided for the network is searched for units not dependent on other units in the network. A call to each function implementing such a unit can be instantly inserted in the resulting C code, and the unit can be removed from the set. At first, this is only true for units connected to an input port of the parent network. Over time there are more and more units whose inputs are connected to an output port of a unit already coded, or to a unit which represents a constant value, a source or sink, or a delay. As soon as all inputs are available, these units may be inserted into the code and removed from the set as well. This search over the set of sub-units is repeated until the set is empty. Then all sub-units are coded in a sequential order preserving the dataflow semantics.

```c
void network_n(state_net_n* unit_state, int* port_a, int* port_b,
      int* port_c, float* port_d, int* port_e)
{
   int ch1;
   int ch2;

   sub_net_x(unit_state->state_sub_net_x1, port_a, port_b, &ch1);
   sub_atm_y(unit_state->state_sub_atm_y1, port_b, port_c, &ch2);
   sub_net_z(unit_state->state_sub_net_z1, &ch1, &ch2,
         port_d, port_e);
}
```

Listing 5.4: Generated network code

The result of this algorithm can, again, be seen in Listing 5.4. If more than one unit is ready to be inserted in the code, these units may be coded in an arbitrary order. For all channels not connected to an input or output port of the parent network, local variables are declared inside the function to allow data exchange between the sub-units. The variables `ch1` and `ch2` are examples for such data buffers. They correspond to the channels connecting `Sub-net x` and `Sub-net z`, or `Sub-atm y` and `Sub-net z`, respectively. For all other channels, the pointers from the function's parameter list are used.

5.4.3 Automata

Just as networks, each automaton is coded as a separate C function. Its parameters are the input and output ports of the automaton. Unlike networks not all elements depicted in an automaton are executed during each invocation. Rather, they are operating in exactly one of their states, depending on the previous as well as the actual input values, and are thus stateful elements. Hence, the signature of an automaton always contains a state parameter. This state parameter is a struct consisting of the automatons currently executed state, represented as an integer value and the state struct for all the automaton's sub-units which implement the automaton's possible different behaviors.

The implementation of the automaton itself is realized as a `switch-case` statement in C code. The switch argument is the reference number of the automaton's currently active state. Inside the according case statement, the guards of all outgoing transitions are computed. If one of the guards evaluates to true, first the

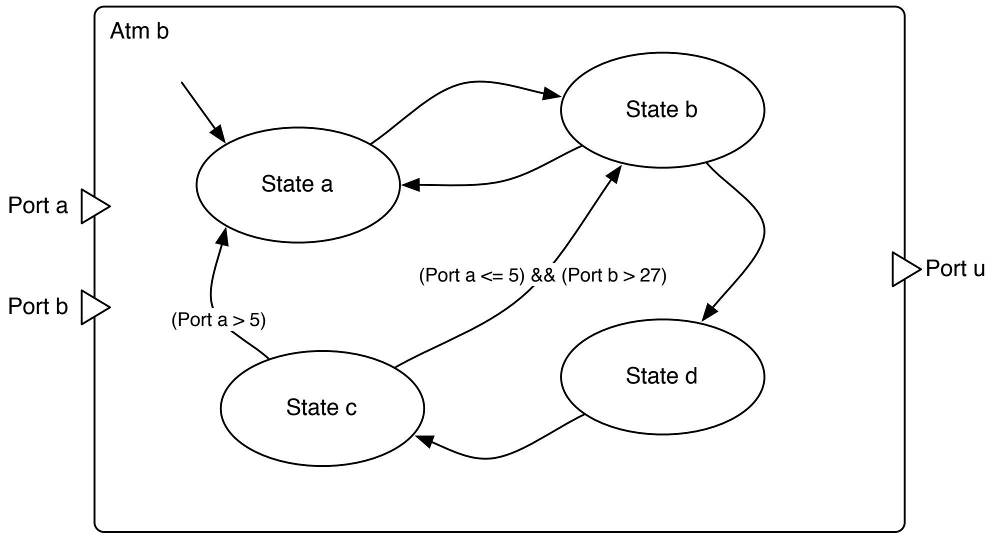


Figure 5.15: An example automaton

reference number for the active state is updated and then the implementation of the target state is called. Otherwise the behavior of the already active state is executed. The example code for an automaton's implementation is given in Listing 5.5. The according automaton is taken from Figure 5.15. In the figure only the guards exemplified in Listing 5.5 are depicted. The Listing shows these guards to be evaluated, if `State c` is active, which corresponds to `case 0`. If any of the guards evaluates to *true*, the variable holding the active state is updated and the according target states is executed. `State c` is otherwise executed. As the units which implement the automaton's states all feature the same input and output ports as the automaton itself, the corresponding parameters of the automaton are passed to the function calls for the implementing units.

In case the automaton is a mode cluster rather than a regular automaton, its implementation is differing slightly. Instead of just calling a sub-unit, a mode cluster decides on a set of other clusters to be executed. To this end it uses facilities provided by the middleware. The details of this approach will be explained in Section 5.6.

```
void atm_b(atm_b_state* unit_state, int* port_a, int* port_b,
      int* port_u) {

   //check active atm state
   switch(unit_state->atm_state)
   {
      //atm "State c" active
      case 0:
         //check first guard
         if(*port_a > 5)
         {
            //update state reference and call implementation
            unit_state->atm_state = 2;
            state_a(&(unit_state->state_a_impl), port_a, port_b,
                  port_u);
            break;
         }
         //check second guard
         if(((*port_a <= 5) && (*port_b > 27)))
         {
            //update state reference and call implementation
            unit_state->atm_state = 3;
            state_b(&(unit_state->state_b_impl), port_a, port_b,
                  port_u);
            break;
         }
         //call actual state implementation
         state_c(&(unit_state->state_c_impl), port_a, port_b,
               port_u);
         break;
      case 1:
         //atm "State d" active
         <implementation omitted>
      case 2:
         //atm "State a" active
         <implementation omitted>
      case 3:
         //atm "State b" active
         <implementation omitted>
   }
}
```

Listing 5.5: Automaton code

5.4.4 Basic Blocks

As has been described before, every network and automaton is encoded as a separate C function. For basic COLA blocks this is not the case. Since basic blocks represent atomic operations like arithmetic or Boolean calculations, calling each operation as a function would induce a huge overhead. Therefore, basic blocks are interpreted inline, which means the according operation is transformed into a single line of code, replacing the basic block with its C counter piece, as one would assume.

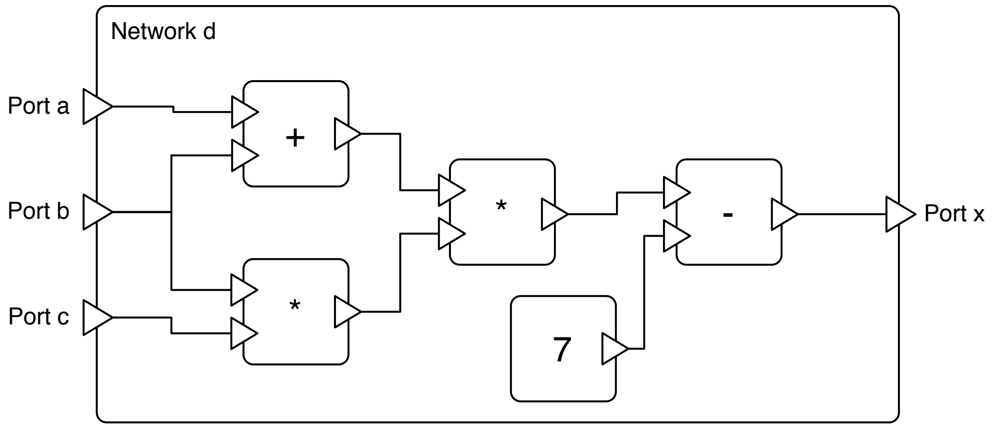


Figure 5.16: A network of blocks

Figure 5.16 shows an example COLA network which consists of a number of blocks. The resulting code is shown in Listing 5.6. As one can easily see, the blocks are replaced by basic C operations. Braces are added to make the precedence of operations clear.

```
void network_d(int* port_a, int* port_b, int* port_c, int* port_x)
{
   *port_x = ((*port_a + *port_b) * (*port_b * *port_c) - 7);
}
```

Listing 5.6: Inlining for a network of blocks

In case of a delay block, the C interpretation would be a simple assignment of the new value to the state struct member holding the delay's value. Thus during code generation, the generator adds a struct member of appropriate data type for each delay to the state struct of the parent cluster.

5.4.5 Code Optimization

It should be obvious that the code generation approach described above results in C code which reflects the behavior of the corresponding COLA model. Following the structure of the model very closely, the use of separate functions for each network in the model hierarchy yields a code that is still readable and can be compared to the modeled system if necessary. Yet, it is not the goal of a code generator for embedded systems to produce the most readable, but rather very efficient code. An exploration of the potential for optimizations has been carried out as student's work by Hierl in [72]. The goals of the work were a reduction of function calls in the resulting code as well as a reduced use of variables, thus speeding up execution and saving the amount of memory needed for the call stack on the target platform.

As another topic the compliance of the generated code to the *Motor Industry Software Reliability Standard* (MISRA) [102] should be examined. MISRA is a common coding standard for high-confidence embedded systems as described by Anderson in [6]. Compliance of the resulting code to the MISRA C standard would be most likely a requirement — as it is for today's automotive software — if the code generator should be certified for use in automotive development. Such a certification would qualify the generator to be used for production code, which could be used without any manual changes in actual cars. This approach would yield a shorter time-to-system, as it would not be necessary anymore to certify every new piece of code. Instead, if the model editor and checker are certified as well as the code generator, any generated code would be considered correct by construction.

All optimizations are based on an intermediate representation of the model as an *abstract syntax tree* (AST). The optimizer parses this tree to search for suboptimal structures, regarding efficiency of the produced code. One optimization of the tree is the so-called *flattening* of networks. Flattening means that the nested hierarchical structure of networks in a COLA model is nullified by reducing all networks to a huge single network with exactly one hierarchy layer. Figure 5.17 gives an abstract example of a hierarchy of networks. Figure 5.18 shows an equivalent flattened network. For reference, the inlined sub-networks are colored in the two figures. As a result of this flattening fewer function calls are produced during code generation. It should be noted that flattening is only used for networks. Whenever an automaton is found in the AST, the automaton is coded as described above. While it would be possible to flatten an automaton too, this might result in huge nested switch-case statements, which are not necessarily more efficient to compute.

Another optimization was realized, targeting a reduction of local variables used for representing channels inside a network. The original version of the code generator would create a separate variable for every channel of a network. The optimization in contrast, reduces the number of variables by reusing them for several channels. To this end, every variable is annotated in the AST with a counter. The counter is incremented when the variable is written, and decremented as soon as the

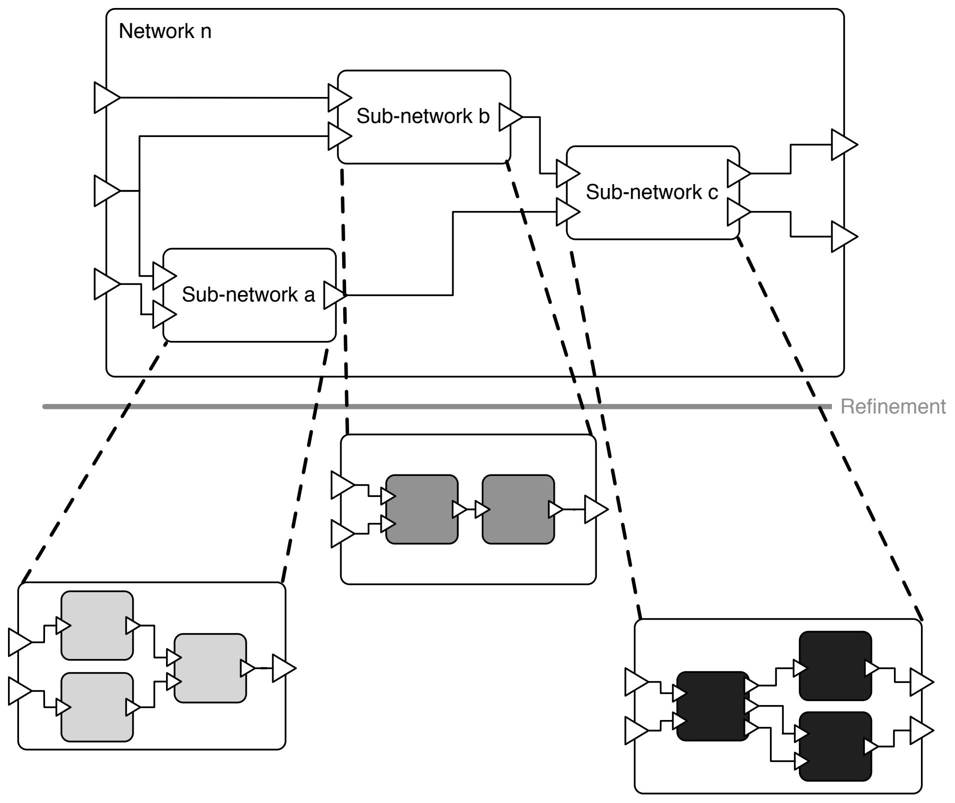


Figure 5.17: The hierarchical `Network n`

according date is used. This is why, the code generator only creates new variables if none with a usage count of zero is left. Otherwise a free variable is used.

The described changes to the code generator have been implemented during the student's work. Tests using several different COLA models showed mixed results. While the binary size of the produced code, as well as the execution speed and stack consumption, where reduced in some cases, other examples showed no change or even reduced efficiency. These non-uniform results point at a dependency on the employed compiler carrying out optimization itself. When using its automatic optimizations, a lot of the additional function calls produced by the original code generator where simply removed by the compiler, that is the compiler used inlining to reduce the number of calls. If, however, optimizations where disabled for guaranteeing equivalence of source code and binary representation, as is state-of-the-art for many embedded systems (cf. [76]), the modified code generator would produce a more efficient result. There is also a lot of effort spent onto proving the correctness of optimizations done by compilers, for example by Engelen et al. [123, 124], Cimatti et al. [35], Necula [104], and others. So maybe in the future, compiler optimizations for production code will be permitted.

Considering MISRA conformance, the work pointed out that the original code

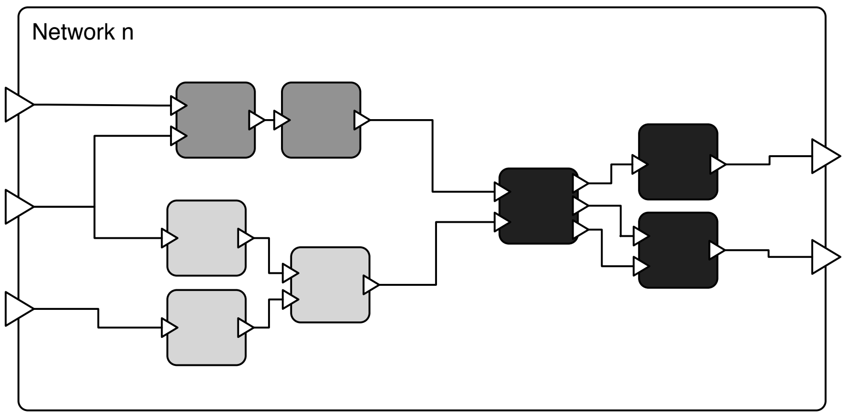


Figure 5.18: A flat version of `Network n`

generator violated several of the rules defined in the standard. However, as Anderson points out in his overview of coding standards for embedded systems [6], only 23 percent of the defined rules are aimed at predictability. These rules shall avoid the use of ambiguous statements of the C or C++ language, whose interpretation is dependent on the employed compiler. All other rules are targeted at simplicity and readability of the code, which is a major concern for human programmers. The analysis of our code generator showed, that it doesn't violate any of the predictability rules, so the resulting software behaves deterministically. As the generated code is not intended to be read or modified manually, the violation of all other types of rules is not a concern and has thus been ignored for the rest of the student's work.

5.5 Allocation and Scheduling

The concepts for allocating and scheduling COLA tasks are joint work with Stefan Kugele, who helped developing the relevant algorithms in this solution. We have introduced the algorithms in detail in [56] and [85], so we will limit the description to an overview of the most important concepts, here.

5.5.1 Allocation

In order to find an optimal solution, it would be necessary to accomplish a combined optimization for allocation and scheduling. But since finding solutions for a valid allocation, or schedule, is complex enough, we kept both steps separate. During allocation the algorithm tries to find a valid mapping of tasks to computing nodes of the system, while the scheduling algorithm aims at calculating a schedule meeting all deadlines for the previously defined allocation. If it is not possible to schedule a given allocation, it is up to the developer to modify the input values for the allocation algorithm and repeat its execution.

To get a guidance for allocation, the algorithm uses *non-functional requirements* (NFRs) specified in the software model and matching the capabilities of the hardware model. The lists of possible requirements and capabilities are not fixed, which means they could be extended in the future according to the respective needs of a specific system. Regarding clusters, the requirements can be separated into essential and auxiliary NFRs. Essential NFRs have to be fulfilled by a valid allocation plan, otherwise the system will not behave as intended. Missing auxiliary requirements do not influence the functioning of the system negatively, but they improve system quality considering safety, cost, etc. For the allocation algorithm, usually both types of requirements are hard prerequisites for a valid allocation plan. If, however, it is not possible to find a valid allocation, the auxiliary NFRs could be removed one after another, until a valid allocation can be calculated. A list of possible essential and auxiliary requirements is given in Table 5.1.

Table 5.1: List of non-functional requirements

Requirement	Unit	Description
cpu_cycles	(ID, cycles)	The amount of processing cycles needed is specific for every processor in question. Thus the value is specified as a tuple mapping the processor ID to a number of cycles.
deadline	ms	Specifies a deadline within a cluster has to be executed.
RAM_req	kByte	The dynamic memory demand during task execution.
ROM_req	kByte	The memory needed for binary file storage.
power_state	Name	Name of the lowest power state in which this task is active.
supplier	Name	The name of the supplier implementing this cluster.
replicas	Instances	The number of copies distributed over the system for redundancy reasons.
cpu_type	Set<Arch.>	The names of valid processor architectures.

The requirements for CPU cycles, required RAM and ROM, as well as the power state a task is active in, are essential. If the computing node, a task is allocated to, does not provide the necessary computing power or memory sizes, the resulting system would not be operational. The power state requirement corresponds to different energy modes the platform might be in. In the following we will give a brief description of the essential requirements:

CPU cycles: Each cluster needs a certain amount of computing power for execution. This amount is annotated to the cluster covering its worst-case requirement. Hence, this requirement can be checked against the given platform. If more clusters are allocated onto a single processing unit than it can handle, not all clusters can be guaranteed to be evaluated.

The value given for the computing power is generated by the SciSim tool presented by Wang et al. [130]. This tool simulates the worst-case amount of processing cycles necessary to execute the code generated for a cluster. This simulation is accomplished for any processor architecture in question, that is each architecture found on any node of the hardware model, which is eligible for executing the task regarding other NFRs.

Deadline: The deadline given for a cluster specifies the maximum amount of time which may pass from the start of a scheduling cycle until the completion of the cluster's execution. If this deadline is not met, the system would not show the desired behavior. The deadline is specific to each application and therefore has to be declared by the developer. During the allocation and scheduling steps, a solution is calculated matching the deadline with the needed computing power for the cluster and the processing capabilities of the platform.

Memory: Similarly to computing power a cluster needs a minimum amount of available memory. Two forms of memory are consumed: first, the binary file generated for a cluster has to be stored in the permanent storage (ROM) of the ECU. Second, the code generated for the cluster has demands regarding the RAM available during execution.

Power state: Typically, embedded systems are restricted to a limited power supply. Huge efforts are put into research and development of power saving technologies. For distributed embedded systems like cars, this can be achieved through the definition of different power states. According to the actual state of the car, e. g. locked, ignition off, ignition on, a varying number of ECUs might be active. Other nodes are shut down at the same time to avoid a waste of power. Allocating a task like the central locking system to such a shut down node would disallow the owner to enter the car again. To avoid such a situation each task should be annotated with its lowest necessary power state.

To distinguish power states, a state hierarchy is given. Each power state defines the set of ECUs running in it. A higher state contains the same, and at least one additional ECU. Therefore the relation $S_0 \subset S_1 \subset S_2 \subset S_3$ indicates four power states which S_0 being the lowest state and S_3 being the highest state.

In addition to the mentioned essential NFRs, we also address auxiliary NFRs. These are not necessary for correct operation, but raise further demands on the system which, for example, lower its cost or improve its efficiency:

Supplier: Large scale embedded systems are often the result of a cooperation of several partners in industry. When defining a model for the whole system, the definition of work packages for the different team partners is desirable. These could consist of several clusters each in case of a COLA system. To allow for this partitioning the designated partner can be annotated to each cluster of the model. The supplier information can then be used to allocate tasks implemented by a single supplier exclusively onto the same ECU(s). This approach enables the partners to retain their current work-sharing where each partner implements a piece of hardware, e. g. an ECU, together with the corresponding software.

Redundancy: Dealing with safety-critical hard real-time systems, demands emerge for the implementation of error correcting techniques in case of a system node's failure. A frequently used technique for error masking is the use of redundant software components, specified by means of clusters in our case. The specification of a redundancy requirement defines the number of redundant cluster copies to use in the system, i. e., on how many different ECUs a cluster should be deployed.

CPU architecture: If a cluster's implementation is dependent on a specific processor's capabilities, e. g. a digital signal processor (DSP), the cluster has to be placed accordingly. This might be necessary for implementations of algorithms requiring a large amount of processing power without violating given deadlines.

Table 5.2: List of capabilities

Capability	Unit	Description
proc_cycles	Cycles/ms	To state the amount of processing power available, the number of cycles per millisecond is given.
cpu_arch	Name	Processing units differ by their respective processor architecture. Thus general purpose processors, DSPs and others can be distinguished.
os_overhead	ms	For every called task, a certain amount of operating system overhead is generated for dispatching, memory management, etc.
RAM_cap	kByte	The working memory available on the node.
ROM_cap	kByte	The amount of permanent memory available on the node.
power_state	State	An ECU is active in the specified power state and all higher power states.
supplier	Name	The name of the supplier building this piece of hardware.

As for the hardware model, a matching set of capabilities can be defined for each ECU of the system. An overview of the defined capabilities is given in Table 5.2. While most capabilities are used as constraints for choosing valid allocation and scheduling schemes, the cost attribute has to be handled differently. It allows for an optimization of the resulting allocation and scheduling plan by calculating the most economic system architecture. We will give a short description of the capabilities in the following:

Processing cycles: For a valid allocation of a task to a processor, the CPU has to provide at least enough computing power to execute the task within its deadline. By specifying the available number of processing cycles per millisecond and comparing them with the amount of cycles necessary for executing the task in relation to its deadline, a valid allocation can be searched. The calculation also takes into account a fixed worst-case amount of time necessary for OS execution (see below), that is, re-scheduling, memory management, and the likes.

CPU architecture: The CPU architectures of all ECUs used by the hardware platform provides input for SciSim, which processors to simulate during worst-case execution time estimation. The allocation algorithm uses the same information for analyzing a possible placement of a task onto an ECU. To this end, the respective number of processing cycles for the task on the CPU in question are loaded from SciSim's list of simulation results.

Further the architecture attribute indicates the availability of a specific instruction set, as it might be the case for a DSP.

OS overhead: A fixed amount of OS overhead is specified for each ECU being required for executing the code of the operating system for scheduling, memory management, etc. This overhead has to be taken into account for each task that is placed onto the respective ECU. Hence, the amount of available CPU cycles is reduced accordingly.

The worst-case figures for the OS overhead have to be evaluated specifically for every platform.

Memory: The available size of permanent and dynamic memory has to fulfill the requirements of all tasks placed onto the respective ECU.

Power state: This attribute specifies the lowest active power state for the ECU. Any task which is active in this or a higher power state, could be placed onto the ECU.

Supplier: If soft- and hardware are provided by the same supplier, this attribute of an ECU should match those of all tasks allocated to it. Other restrictions like a co-operation between different manufacturers could also be defined by using a matching label for the soft- and hardware resulting from the co-operation.

The stated requirements and capabilities are used by the allocation algorithm to find a solution for the allocation problem. Its evaluation is resolved using *integer linear programming* (ILP), akin to the approaches by Matic et al. [99] and Zheng et al. [134].

In order to optimize the allocation for a given metric, costs may be assigned to all properties. The allocation algorithm then tries to minimize the cost function during allocation. For the exact algorithm, please refer to [85]. The allocation result is stored in the COLA model and by this made available to the scheduling tool.

5.5.2 Scheduling

Based on the result of the allocation step, the search for a suitable schedule can take place. For our scheduling approach we rely on *time-triggered co-operative* (TTC) scheduling [91]. In a time-triggered co-operative system, tasks are scheduled offline, that is, before the system is executed, according to their respective worst-case execution times, frequencies, and deadlines. Preemption is not allowed in this scheduling scheme, hence the term co-operative. The frequency of repeated execution of a task as well as its deadline have to be given by the COLA model. Its worst-case execution time is calculated automatically by the SciSim tool. Using these data an overall system schedule, with respect to data dependencies, can be derived. As the schedule is dealing with control loop applications, it is executed cyclically, according to the task frequencies. At the beginning and end of each scheduling loop, the sensors are read and the actuators written, respectively. By that the model assumption of values read and written from and to the environment at the same instant in time can be best imitated.

Figure 5.19 shows an example of a possible scheduling loop. In the figure, a group of `n` ECUs is executing an application of eight tasks. At the beginning of the cycle sensors are read, then the tasks `t1` to `t8` are executed and finally, actuators are written. The execution of the tasks is distributed over the ECUs, thus requiring communication between the ECUs. This exchange of message is depicted by the small arrows between the tasks. In order to guarantee the availability of the communication medium, the bus is regarded by the scheduling algorithm as an additional ECU. Accordingly, the scheduler assigns not more than one message to the bus at any time, thus avoiding collisions. Messages transmitted over the bus are indicated by the small gray boxes in the lower part of Figure 5.19. Each box corresponds to a message exchange of the tasks shown in the upper part of the figure. The resulting schedule for the bus can subsequently be used to define an appropriate TDMA scheme for bus communication.

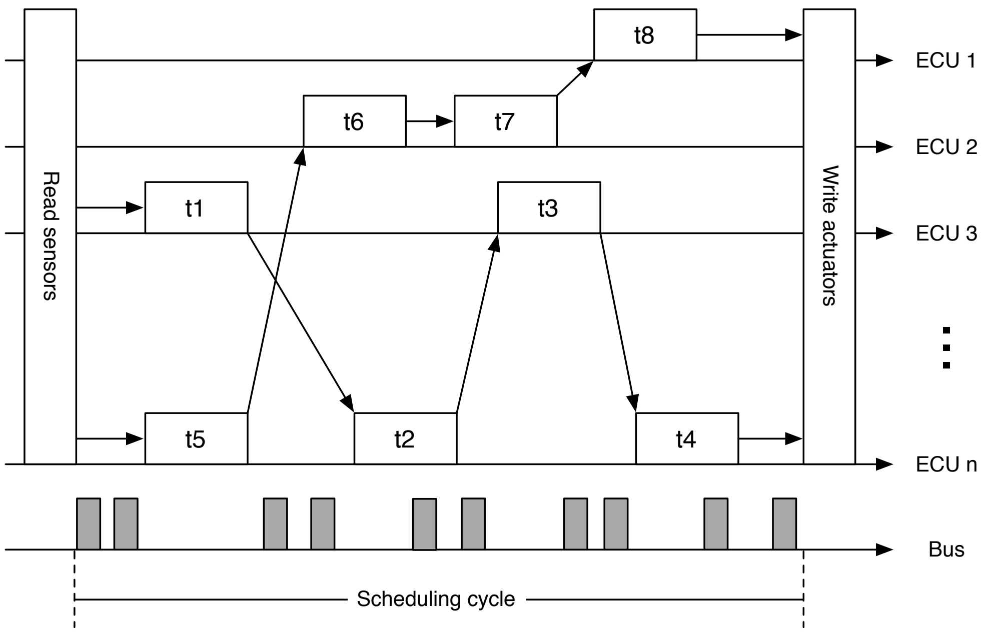


Figure 5.19: Scheduling cycle

For improved readability jitter and overhead caused by clock drift, operating system, and middleware execution are omitted in Figure 5.19. The according worst-case assumptions are added to each task's WCET and each inter-task communication.

The CDG serves as an input for the scheduler during calculation. Using the contained information about data dependencies between the clusters, a valid order of tasks can be chosen. Based on the number of mode cluster vertices contained in the graph and the number of their related sibling nodes, the scheduler is able to calculate the number of different schedules to calculate. These schedules are necessary for implementing the system operating modes, as we will describe in the next section. For the exact algorithm of the scheduler, please refer to [56].

5.6 System Operating Modes

As mentioned in Section 4.3.4, COLA allows for the definition of operating modes. Each mode cluster triggers a set of subordinate clusters for execution. During code generation all clusters are transformed into executable tasks. Hence, the activation of a mode, and all of its sub-modes, is calculated by a task as well. We call such a task a *mode-task*. Like all other tasks, called *working tasks* in the following, mode-tasks use the communication abilities of the middleware. Compared to working tasks, mode-tasks do not write any actuators directly, but output a value, which indicates their decision on the mode to be activated.

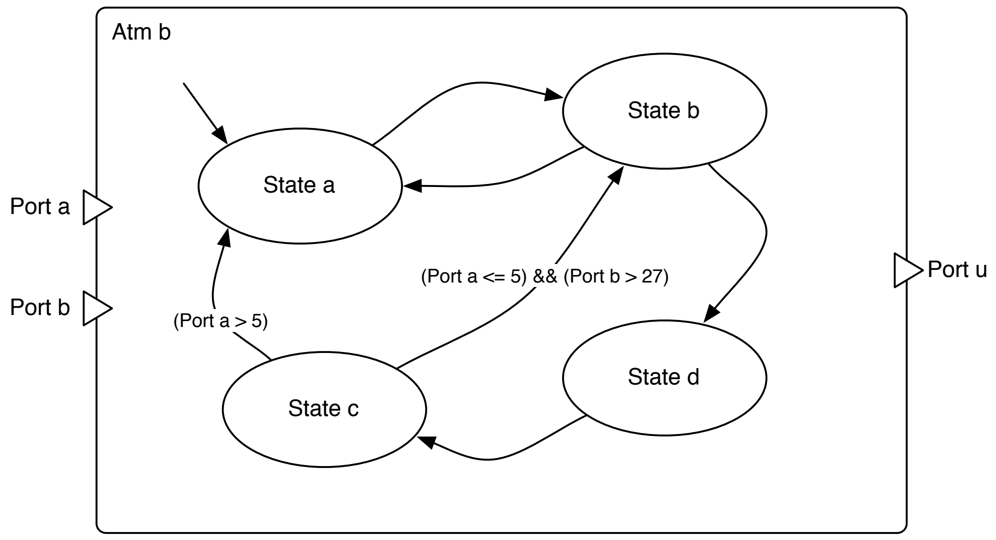


Figure 5.20: An example mode automaton

5.6.1 Generating Code for Mode Clusters

According to their definition, we require mode clusters to consist solely of automata. Hence the code generated for a mode cluster is similar to that of an automaton. But instead of calling a function which implements the automaton's active state, a value indicating the current operating mode is written to a distinct middleware address. The scheduling routine reads this value to decide which subsequent tasks to execute. These tasks may again be mode tasks, resulting in a hierarchy of operating modes, or working tasks which implement the current mode.

We reuse the automaton shown in Section 5.4.3 to provide better comparability of the resulting code. Assuming that the automaton from Figure 5.20 was not contained inside a working cluster, but rather in a mode cluster, the code shown in Listing 5.7 would be generated. Listing 5.7 shows just the implementation of `State c`, but the remaining states would be implemented analogously.

Differing from the code shown in Section 5.4.3, the code for a mode automaton contains a local integer variable `mode`, which can be seen in line 3 of Listing 5.7. This variable is used as a local buffer for the calculated mode, before writing it to the middleware. Since we are using a mode cluster here, the cluster's state as well as its inputs are read from the middleware as shown in lines 6 through 8.

If the automaton is invoked with `State c` active, the conditions of the outgoing transitions are checked first. In line 15 the first condition is evaluated. If the result is `true`, the active state changes to `State a`, which corresponds to an integer value of 2. Hence, this value is set as the new automaton's internal state in line 18, as well as assigned to the buffer as result in line 19.

```c
void atm_b() {
   atm_b_state unit_state;
   int mode, port_a, port_b;

   //get actual state and input values
   mw_restore_task_state(&unit_state, sizeof(unit_state), 17);
   mw_receive(&port_a, sizeof(port_a), 22);
   mw_receive(&port_b, sizeof(port_b), 23);

   //check active atm state
   switch(unit_state.atm_state)
   {
      //atm "State c" active
      case 0:
         if(port_a > 5)   //first guard
         {
            //update state and result variable
            unit_state.atm_state = 2;
            mode = 2;
            break;
         }
         if(((port_a <= 5) && (port_b > 27)))   //second guard
         {
            //update state and result variable
            unit_state.atm_state = 3;
            mode = 3;
            break;
         }
         //update state and result variable
         mode = 0;
         break;
      case 1:
         //atm "State d" active - implementation omitted
      case 2:
         //atm "State a" active - implementation omitted
      case 3:
         //atm "State b" active - implementation omitted
   }
   //write mode and updated state to middleware
   mw_send(&mode, sizeof(int), 31);
   mw_save_task_state(&unit_state, sizeof(unit_state), 17);
}
```

Listing 5.7: Mode automaton code

Equally, the second guard is evaluted from line 22 on, if the first transition has not been taken. Finally, if none of the transitions is taken, the current state remains active and thus the `mode` variable is assigned `0`, which is the integer value corresponding to `State c`.

After the automaton is fully evaluated, the result value buffered in the `mode` variable is written to the middleware for reference by the local scheduler. The according middleware call can be seen in line 40. Finally, the updated state is stored to the middleware in line 41.

5.6.2 Operating Mode-Aware Scheduling

As explained earlier, the result produced by a mode task triggers a set of other tasks for execution. Accordingly, the current schedule has to be altered to disable the old and enable the new set of tasks to be executed.

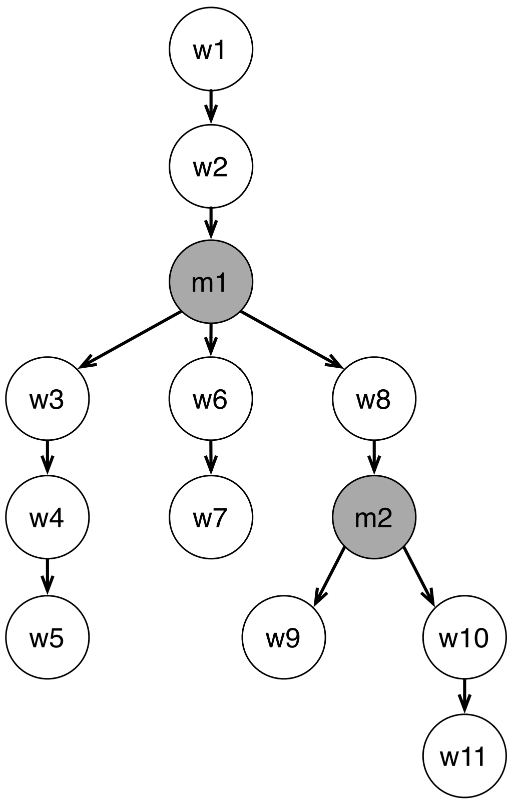


Figure 5.21: Scheduling graph

To this end, the scheduling tool described above is aware of mode clusters in the model and considers a branching after each mode cluster inserted into the schedule. Figure 5.21 points this approach out. Mode clusters are marked grey in the figure, while working clusters are white. At the beginning of the scheduling cycle, some working clusters might be scheduled, see `w1` and `w2` in the figure, before at some point a mode cluster is executed, `m1` in our example. Depending on the number of states of the mode cluster, a corresponding number of possible sub-schedules might be executed thereafter. In the example shown in Figure 5.21, mode cluster `m1` features three states allowing three possible sub-schedules to be triggered. Each of these sub-schedules may again contain working clusters, `w3` through `w11` in the figure, or mode clusters, in our example `m2`. This requires a branching into several sub-schedules. As a result the overall system schedule is a directed graph, as shown in Figure 5.21. For each path from the root node to any of the leaf nodes, a scheduling plan has to be calculated by the scheduling tool. Figure 5.22 shows the derived schedules `S1` through `S4` for the scheduling graph from Figure 5.21.

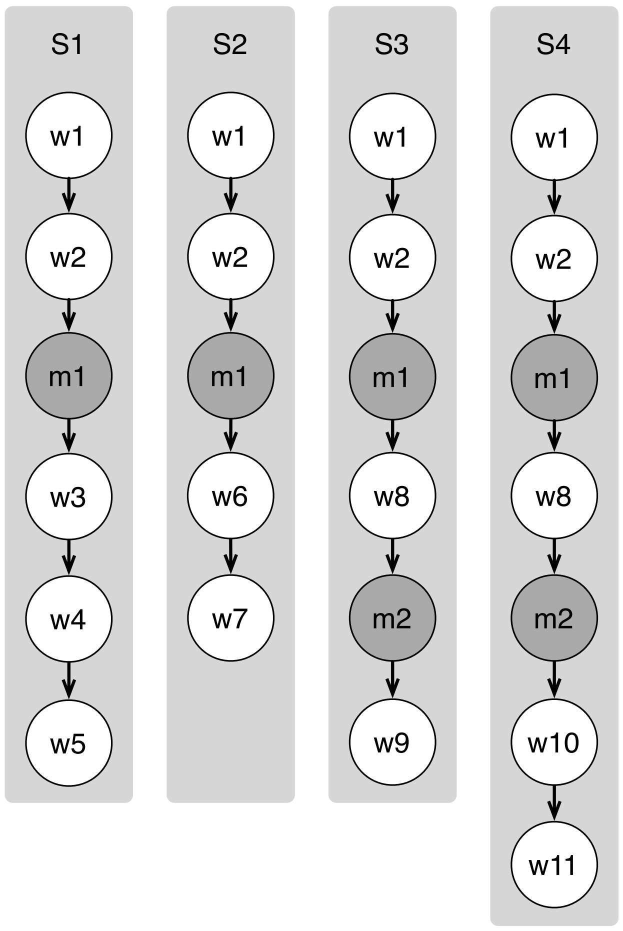


Figure 5.22: Scheduling options

The actual switching of tasks is carried out by the scheduling loop, the generation of which we will explain in the following section. Basically, each time a mode task has finished execution, the scheduler reads the result from the middleware.

Subsequently, the scheduling loop calls the according set of tasks. Since the schedule is calculated system wide, and its execution is time-triggered according to a globally synchronized clock, it can be guaranteed that the change of an operating mode will be accomplished system wide and reliably. To achieve this, all tasks depending on the mode task's decision are not started by the respective scheduling loops, before the mode task is fully executed and its result communicated throughout the system by the middleware.

5.7 System Configuration

After functional code for the COLA model has been generated, and allocation as well as scheduling are determined, the software is ready to be executed on the target platform. This demands the configuration of the basic software, that is, operating system and middleware, according to the generated allocation and scheduling schemes. The middleware is handed over the specific addressing scheme, including the generated logical addresses from the CDG, the timing for message transmission and reception, and the polling of respective sensors and actuators. Likewise, the operating system is configured with the timing calculated for task execution.

Similar approaches for the configuration of automotive platforms have been proposed by the *OSEK Implementation Language* (OIL) and AUTOSAR. OIL differs from our approach in matters of its target systems. The attributes described in OIL aim at configuring the scheduling of a single node, not that of multiple ECUs — let alone their communication. Descriptions of OIL's features can be found in the related standards document [105] as well as an article about OIL by Zahir [133].

AUTOSAR on the contrary is aimed at distributed systems. Generation of the information captured in an AUTOSAR model are transferred to a specific platform by means of the RTE, which is responsible for timely task execution and communication in the system. Compared to the deployment approach described here, this configuration is transformed into hardcoded C code statements. A subsequent change of the configuration, say a change of logical addresses, is therefore only possible by re-generating the entire code — a drawback AUTOSAR shares with OSEK. Since our middleware loads its configuration from a config file during each start-up, it is much more flexible regarding such changes.

We will give an introduction into the configuration parameters for our middleware as well as the task scheduler in the following.

Middleware settings. We will explain the relevant settings for our middleware, using the example given in Listing 5.8. The example is in XML format, though an implementation for a productive automotive system would employ a more memory efficient format for data storage. However, the use of XML was also used with our prototypical implementation of the middleware, as it is easier to understand (and debug), and we will stick to it here for the same reason.

The first information in the middleware configuration is related to the length of the communication cycle and the slot definition for that cycle. In line 2 the length of the TDMA cycle is specified by the attribute `cyclelen`. Further, the `id` of the operating mode is given. For every operating mode, such a definition of slots is necessary. Next in lines 3 through 9, the communication slots of the TDMA cycle in that operating mode are enlisted. Each slot definition states the numerical slot id and the respective starting time in milliseconds relative to the start of the current cycle.

```
 1  <mw_config>
 2     <mode id="0" cyclelen="210">
 3        <slot id="0" time="0"/>
 4        <slot id="1" time="5"/>
 5        <slot id="2" time="12"/>
 6        <slot id="3" time="27"/>
 7        <slot id="4" time="34"/>
 8        <slot id="5" time="55"/>
 9        <slot id="6" time="63"/>
10        ...
11        <slot id="18" time="200"/>
12     </mode>
13     ...
14     <node id="0" master="1">
15        <sensor slot="" dataid="122" datalen="3" driver="/dev/rfcomm1" cycle="1" init="0"/>
16        <sensor slot="14" dataid="130" datalen="3" driver="/dev/rfcomm2" cycle="1" init="0"/>
17        <actuator slot="" dataid="135" driver="/dev/motor1" cycle="1" init="50"/>
18        <task id="106" slot="2" name="normal206613" trace="0">
19           <data id="137" channel="in" name="distance_front" type="INT" len="4"/>
20           <data id="103" channel="out" name="parking_ready_out" type="INT" len="4"/>
21           <data id="101" channel="out" name="emergency_stop_out" type="INT" len="4"/>
22        </task>
23        <task id="108" slot="3" name="speed_control" trace="0">
24           <data id="128" channel="in" name="speed_in" type="INT" len="4"/>
25           <data id="134" channel="out" name="speed_out" type="INT" len="4"/>
26        </task>
27        <task id="104" slot="12" name="rotation_count" trace="0">
28           ...
29        </task>
30        ...
31     </node>
32
33     <node id="1" master="0">
34        ...
35     </node>
36
37     <node id="2" master="0">
38        ...
39     </node>
40  </mw_config>
```

Listing 5.8: Middleware configuration

Following the TDMA scheme, the configuration for the different nodes of the system are given. Lines 14, 33, and 37 show the particular beginning tags for different `node` elements of the XML. Each node refers to an ECU of the target platform. It is important to understand, that all nodes of the system receive the same configuration file containing the sending slots of all adjacent nodes. This information may be used by a middleware instance to determine that point in time, or more precise the TDMA slot, when input data for a local cluster can be received from the bus.

The `node` tags include a reference number related to the COLA hardware model and a Boolean value called `master`. This value declares whether the relative node is the bus master, being responsible for sending global time synchronization messages, or not. This information as well as the TDMA scheme is passed on to the bus driver by the middleware. Inside each `node` element the sensors and actuators connected to the node are specified, as well as the tasks allocated to it. While the information about hardware components is, again, taken from the hardware model, the allocation is the result of the afore described allocation algorithm.

In the example two sensors and one actuator are connected to the node with `id 0`. For each of these components, a TDMA slot may be specified, as can be seen in line 16. Accordingly, the data received from the sensor are transmitted over the network using the specified slot and may be used by a task on another ECU. If no value is specified, the data are only used locally. The `dataid` defines the logical middleware address for the particular value. The middleware instance reserves a buffer of sufficient size, accordingly.

As explained in Section 5.2, we rely on the hardware components to feature a UNIX-like driver interface. The `driver` attributes in lines 15 through 17 tell the middleware which device to open for interfacing the respective sensor or actuator. The polling frequency of each component is declared in multiples of TDMA cycles, using the `cycle` attribute, and the `init` value specifies initial values for safety reasons for each piece of hardware, though these values are not necessary if the system operates normally. For sensors an additional parameter named `datalen` specifies the size of the datum to read.

Following sensors and actuators, the tasks allocated to the particular ECU are declared. In Listing 5.8 three tasks are defined in lines 18, 23, and 27. The `id` of each task is identical to the address defined during dependency analysis in the CDG. Thus, the task may store its internal state using this address. The `name` attribute is meant for debugging purposes only, while the Boolean `trace` attribute specifies, whether the task's internal state, input, and output shall be recorded during execution. This flag can be set in the model editor and the saved information may be used for model-level debugging, which we will introduce in Section 6.1.

Inside each task element its communication points to the middleware are defined. This information is extracted from the communication channels of the according clusters in the COLA model. Each of the `data` elements carries an id identical to the respective buffer vertex of the CDG, a direction whether it specifies an in- or output datum, a data type and the size of that data type. For debugging purposes, again, the name of the original COLA channel is mentioned. The described scheme is repeated for each tasks and ECU.

Using this configuration file, the middleware is able to send and receive messages in the pre-defined slots of the TDMA schedule and to allocate an appropriate amount of memory for all exchanged data.

Scheduling plan. For the scheduling of tasks on each ECU, a configuration file with the scheduling plan calculated by the afore described scheduling tool is generated. As mentioned, the particular scheduler instance, which is implemented in the operating system of each ECU, is not required to make any scheduling decisions at runtime. It serves rather as a dispatcher for activating tasks according to the schedule, which has already been constructed offline.

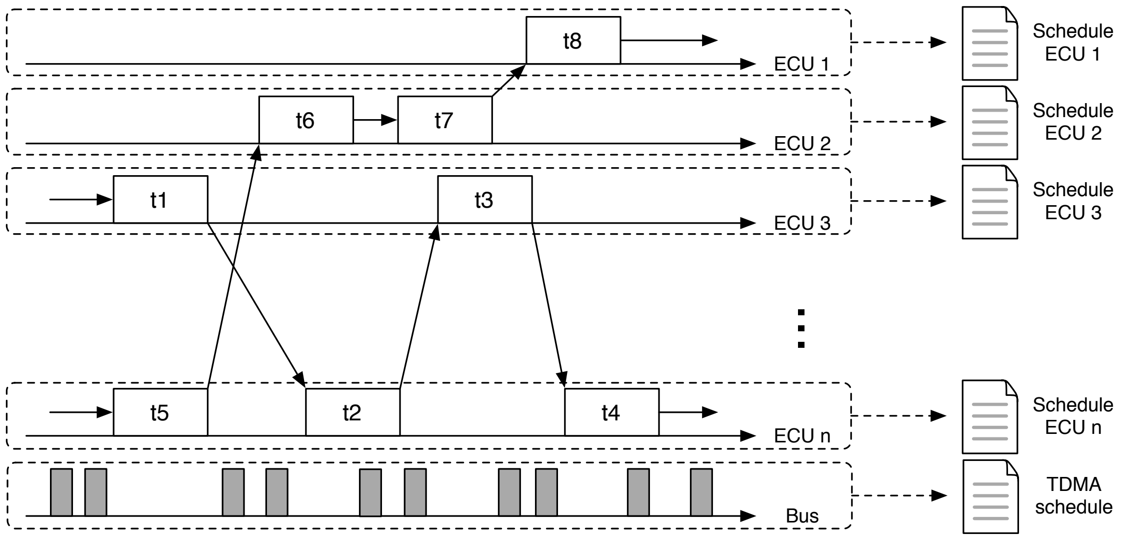


Figure 5.23: Construction of per-ECU schedules

Figure 5.23 shows the graphical representation of a possible output of the offline scheduler. The example corresponds to the example system already presented in Figure 5.19. As can be seen in the figure, the resulting schedule is an overall system schedule. Thus, all ECUs contained in the system are depicted with their respectively allocated tasks. The bus used for communication in the system is shown as another resource, being busy during message transmission. The depicted schedule is repeated cyclically according to a defined frequency.

As indicated in Figure 5.23, the configuration for individual ECUs is derived from this system schedule. To this end, a code file containing only the tasks allocated to the ECU in question is generated. Listing 5.9 represents a code snippet from the scheduling loop for a single ECU. In line 4 the scheduling function for an ECU is defined. It's first action is to wait for synchronization of the global time carried out by the bus protocol. As soon as the ECUs are synchronized all ECUs, as well as the bus controllers, execute the defined scheduling cycle, as can be seen in line 8.

```
 1 TIME time_start, time_actual, time_sleep;
 2 const TIME length_of_cycle = 210;
 3
 4 function schedule_loop() DO
 5
 6    //wait for global time "0"
 7    wait_for_initial_sync();
 8    //start operating system loop
 9    while(true) DO
10       //global time at start of the loop
11       time_start = mw_global_time();
12
13       //get recent time and wait until 12ms
14       time_actual = mw_global_time();
15       time_sleep = 12 - (time_actual - time_start);
16       if (time_sleep > 0) DO
17          sleep(time_sleep);
18       DONE
19       task_id106();
20
21       //get recent time and wait until 27ms
22       time_actual = mw_global_time();
23       time_sleep = 27 - (time_actual - time_start);
24       if (time_sleep > 0) DO
25          sleep(time_sleep);
26       DONE
27       task_id108();
28
29       ...
30
31       time_actual = mw_global_time();
32       //wait till end of cycle
33       time_sleep = lenght_of_cycle - (time_actual - time_start);
34       if (time_sleep > 0) DO
35          sleep(time_sleep);
36       DONE
37    DONE
38 DONE
```

Listing 5.9: Per-ECU scheduling loop

At the beginning of each cycle the actual global time is stored to a variable. The code example comprehends an appropriate middleware call in line 11. To calculate the waiting time until the next execution of each task, the actual time is read again, as shown in line 14, calculating the offset to the beginning of the current cycle, which is exemplified in line 15. Subtracting this offset from the desired starting time of a task gives a slack time, defined as a sleep period for the scheduling. An example of this is presented in line 17. Finally, after the slack time has passed, the task is executed as shown in line 19.

This scheme is repeated for every task that has to be executed on the respective ECU. After the last task has finished, the remaining time in the current scheduling cycle is calculated, as can be seen in line 33. To this end the cycle length specified in line 2 is employed. After sleeping for the rest of the cycle, the scheduling loop is repeated thus beginning the next invocation of tasks.

In the given example, we omitted the use of an operating mode to avoid confusion. If operating modes are present in a system, the scheduling loop is extended with calls to the middleware. These calls are used for receiving mode decisions from mode clusters. Depending on the respective result, the scheduling loop chooses the appropriate set of tasks for execution.

5.8 Chapter Summary

In this chapter we presented our concepts for deploying an automotive system modeled in COLA onto an actual distributed target. By making use of the entire amount of information provided by an integrated modeling language, we are able to not only generate code for single functions, but also calculate an allocation, a valid scheduling scheme, and configuration data for the target platform.

This approach is facilitated by defining standards for driver interfaces and scheduling behavior of the operating system, along with a custom middleware comprising key features like transparent communication and sensor polling, which have to be provided by the target platform. The result is a fully integrated and ready-to-run system, which adheres to hard real-time constraints and follows the model's semantics as closely as possible.

Besides the basic requirement of generating a properly functioning system, the information captured in COLA can be used during deployment to produce even higher quality systems. In the next chapter we will show how runtime information can be mapped back to elements in the model editor for debugging, and how fault-tolerant operating modes can be generated automatically. Finally, a possible integration of certain aspects of the deployment concept into an existing AUTOSAR development process is outlined.

Chapter 6

Extension of the Deployment Concept

In the preceding chapters, we presented the use of COLA for the specification of embedded systems enabling the unattended generation of code and configuration data for a distributed automotive platform. Besides this primary goal, our approach allows for further improvement of system quality and higher flexibility. In this chapter we will present two concepts for increasing the reliability of systems, as well as a concept for the deployment onto an alternative platform.

The first concept presented here is aimed at further reducing the number of faults in the generated system. While the absence of syntactical programming faults can be assured by the code generation tools, there is still a possibility for functional errors due to a faultily designed COLA model. To enable the detection of such faults in the model, we introduce the concept of model-level debugging in Section 6.1.

Improvement of the system's quality can also be achieved by providing concepts for its safe operation even in the presence of hardware failures. In Section 6.2 we propose an extension of our deployment concept allowing the generation of fault tolerance modes. These modes are based on a redundant allocation of highly safety-critical tasks. As a result the defect of an ECU of the target platform my be masked or at least a graceful degradation of the system is achieved.

The third concept presented in this chapter deals with the integration of functionality modeled in COLA into existing AUTOSAR systems. In principle, the modeling in COLA is intended to cover entire automotive systems. Only if the whole system is specified in COLA and deployed to a suitable platform, full advantage can be taken of its integrated approach. However, since AUTOSAR is already used in actual automotive systems, a method for integrating hybrid systems, consisting of tasks modeled in COLA alongside tasks modeled in AUTOSAR, is beneficial. This would allow for a step to step migration from an AUTOSAR- to a COLA-based development process. The necessary changes to the COLA deployment concept for generating such hybrid systems are described in Section 6.3.

The implementation of the three concepts is facilitated by two key factors of the COLA approach: first, the model covers the necessary software and hardware characteristics to generate complex distributed systems. The availability of these information leads to a comprehensive view of the system, and opens the possibility to address complex, system wide issues. Second, the COLA language abstracts from specific implementation details like programming language, operating system, and employed hardware. Thus, the same COLA model — or parts of it — may be deployed onto different platforms by adapting the code generation tools for the relevant platform.

6.1 Model-Level Debugging

As explained in Section 4, the complexity of a designed real-time system calls for an automatic transformation of the model into executable code. The tools built around COLA not only enable the generation of code representing the modeled functionality, but also accomplish the configuration of the distributed target platform. This automated translation reduces the number of faults and guarantees reproducible results, while improving overall quality.

But even such a generated system may show failures due to faults contained in the system specification, or because of limited knowledge about the exact environmental conditions. Consequently, faults become evident not until executing the software on the target platform and are hard to debug regarding its embedded nature. To get a grip on such faults calls for a technically mature debugging concept, even in a MDD process.

In order to deal with the problem, feedback from the executed system to the model would be desirable, but recent MDD concepts do not provide this feedback. Instead, developers typically start to change the model on a trial and error basis or — due to a lack of time — ignore the model altogether and commence to debug at source code level. This practice could be prevented by providing adequate tool support for debugging the model.

While code generation for dataflow models is well known [18], we are not aware of any approach facilitating the debugging of a generated system therefrom at model-level. Regarding safety requirements for generated code as well as its readability, we consider this form of debugging a must. The approach has been first presented in [54]. Since existing concepts like RT-Simex [39] or Rational Rose RT[1] are based on the UML, the exact functional implementation of a system may not be modeled. Hence, the traced application data can only be mapped to the respective structural model, giving just the approximate location of the error.

6.1.1 Sources for Errors

In current MDD practice systems development is often intended to be a one-way process. The system is designed by means of models. Simulation is used to validate these models against the functional specification. If no more errors show up during simulation, code is generated from the models which is then tested on the target platform. In our experience, however, it often happens that the code does not behave as intended, because some types of errors cannot be identified during static analysis or simulation of the model. These system errors may arise for different reasons:

Environmental feedback: One reason might be faulty assumptions about the actual system behavior, which are not discovered during simulation at model-level because of missing or incomplete environmental feedback. For example, if an adaptive cruise control system adjusts the current throttle position to increase the speed of a vehicle, the wheel speed sensors will subsequently deliver higher values due to the increased velocity. But these values are also influenced by the weight of the vehicle, the current wind speed, etc. Including all these parameters in simulation demands a complex environmental model, which is time consuming and difficult — if in some cases not impossible — to provide. Therefore, the initial design of some applications is more or less an approximation of the final system, and only tests on the target platform allow to verify the correct functioning of the design.

Digitalization errors: Another source for unintended system behavior may be suddenly varying sensor and actuator accuracy. A sensor will eventually deliver values different from assumptions made in the model, due to digitalization errors. Unexpected, intermittent steps in the delivered values are a prominent example of this problem. Similarly, actuators might react differently or less accurate than anticipated to rapidly changing real-world values or ambient conditions.

Erroneous specifications: Finally, the generated system is only as good as the underlying model. During development of a large scale system like an automobile, some information about the characteristics of employed hardware

[1]http://www.ibm.com/software/rational

components might not be available right from the beginning. Rather, occurring communication delays, overhead, processing speeds, and accuracy of the actual hardware components might differ from the specification stated in the hardware model. Such deviations might occur when pieces of hardware are replaced by an alternative, not entirely identical choice. Or, for example, a piece of hardware behaves slightly different than specified in the associated data sheet because of a spread for standard factory models.

Since the mentioned sources for errors are also valid for MDD, debugging is still an elementary task in such a process.

6.1.2 Classical Debugging

Figure 6.1 shows a comparison of classical debugging on the left, and the new approach of model-level debugging on the right. Without appropriate tool-support for debugging at model-level today, developers start to debug the generated code — even in a MDD process. Thus classical debugging is carried out at platform-level. In a MDD however, we propose to carry out changes solely at model-level. This can be achieved be employing the concept shown on the right side of Figure 6.1. There, the generated code is executed on the target platform and runtime data of the system in execution are captured. These data may then be mapped back to the model to identify problems at model-level, rather than at code-level.

Model-level debugging is favorable in comparison to classical debugging for several reasons:

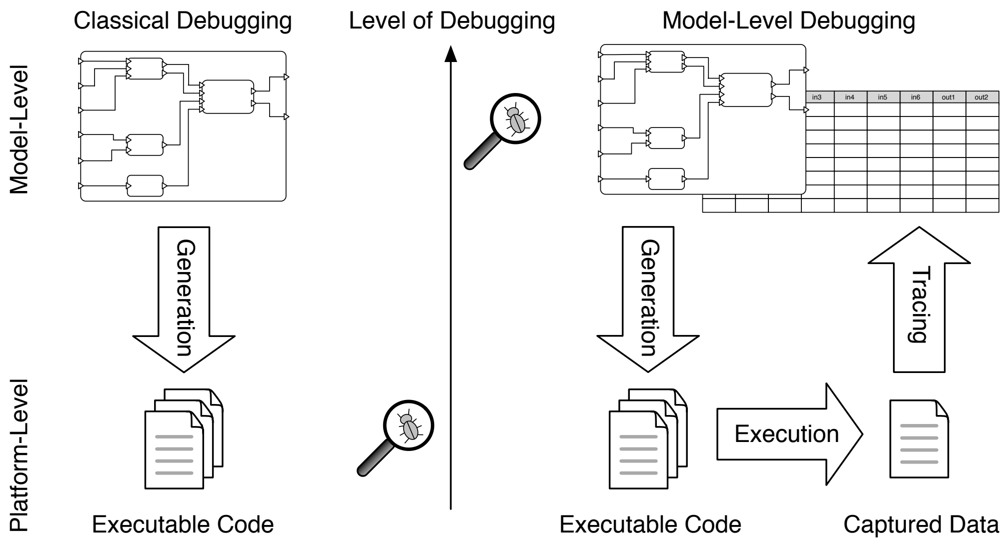


Figure 6.1: Classical versus model-level debugging

- In embedded systems development there are very limited possibilities for communicating debug information. Textual error messages — as oftentimes used with desktop systems — cannot be employed typically due to the lack of a display. Instead, debug messages are signaled by a blinking LED or other low-level communication mechanisms, making debugging much more time consuming. A mature debugging approach for embedded systems should provide a concept for reporting errors more conveniently. Existing debuggers like the GNU Debugger[2] or the Trace32 In-Circuit Debugger[3] typically provide raw data. They can capture datagrams transmitted by a network or bus system, or save the stack and register contents of a processor during execution on an actual CPU. Few of the tools provide a mapping of these values to the source code — let alone to a model — of the executed application. The gathered results are rather shown as lists of values. Our new approach, however, is aimed at the debugging of applications at model-level using data captured at the target system. This new debugging method closes the gap between graphical model and executed code.

- An additional problem, especially when employing MDD, arises from compliance requirements regarding source code which is automatically generated from the model. If a certified code generator is used — i.e., it is proven that the generator retains the model semantics — the code is assumed to be correct with respect to the model. Any manual modification of the generated code must be prevented in order to preserve its correctness. Thus changes to the design, if necessary, should be carried out at model-level.

- Finally, during implementation of a new system, no hardware prototype might be available for testing the corresponding software design. However, designing a high quality software model for the system calls for simulation, and even software-in-the-loop (SIL) tests, based on realistic input values. Regarding our target domain, namely automotive systems, the designed components are frequently not entirely new concepts, but enhancements of existing ones. This is why it would often be desirable to use values captured on the previous version of a component for an initial test of the new design. Consequently, a debugging approach for these domains should provide a solution to this requirement. Using the approach described here it would be possible to use previously stored input data for simulation of newly modeled systems. Similarly, those values could be used as inputs for hardware-in-the-loop tests, facilitating black box testing of combined hardware/software components.

To simplify the debugging of systems modeled in COLA, we propose a new approach for a debugging at model-level. Although it is based on COLA's basic

[2]http://www.gnu.org/software/gdb
[3]http://www.lauterbach.com

concept of a comprehensive system model, we believe the approach to be applicable to different modeling languages, if the availability of the necessary information about soft- and hardware can be achieved.

6.1.3 Model-Level Debugging Concept

As indicated on the right-hand side of Figure 6.1, our model-level debugging approach aims at elevating the troubleshooting to the model-level. Using COLA, a system model is defined and subsequently transformed into executable distributed code by our code generator. The middleware is used for interaction between the generated tasks on the execution platform. That means all communication of a task is routed through the middleware. Additionally, the middleware serves as a storage for the task's internal state. Thus the middleware is able to capture input and output values as well as the task's internal data for later debugging. Since data are always routed through the middleware, there is virtually no impact on the timing behavior of the system. If the tiny deviation in timing is relevant for an application, the constant overhead could be easily compensated by the schedule.

During system modeling, the developer is assisted by COLA's model-level simulator [71]. To start a simulation, the developer can specify arbitrary input values for the system using its graphical user interface. If input data of a real system, like the ones logged by the middleware, are available, a more realistic replay can be accomplished. The designed system model, as well as the corresponding values captured at platform-level, can be loaded into the simulator. These real-world values are then used for simulation. Compared to simulation based on fictitious values, the replay of data acquired at platform-level exhibits realistic system behavior. Realistic debugging at model-level is made possible this way. To trigger the logging of runtime data, the developer may set a flag for each cluster whose data shall be captured. This information is passed to the middleware instance on each node of the system by means of the system configuration file. Thus, the middleware instances know which data to store during execution.

The described approach solves the previously mentioned difficulties. Thanks to the import of captured data into the simulator, a convenient display of runtime data is achieved. All values are shown next to the corresponding model elements, allowing quick and easy identification of design faults as well as their elimination. Changes are completely performed at model-level, not affecting code integrity. Finally, the acquired input values can be reused for future simulations of new designs, without the need for a hardware platform available from the very beginning.

6.1.4 Realization of Model-Level Debugging

Figure 6.2 gives an overview of the necessary steps conducted during the tracing process.

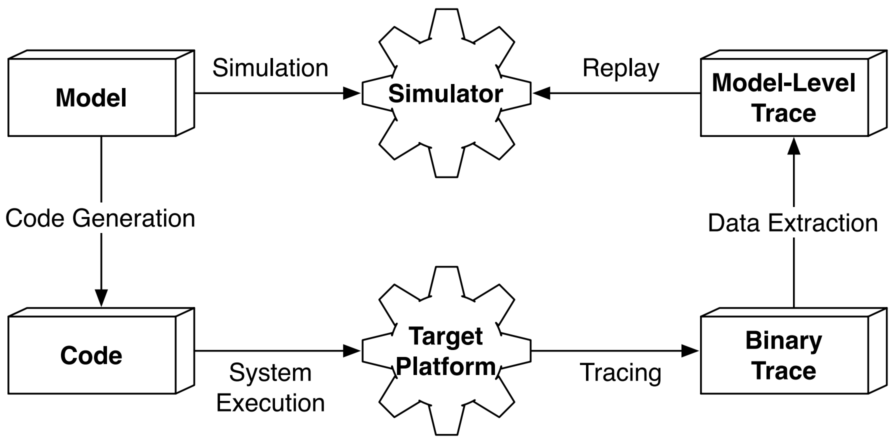
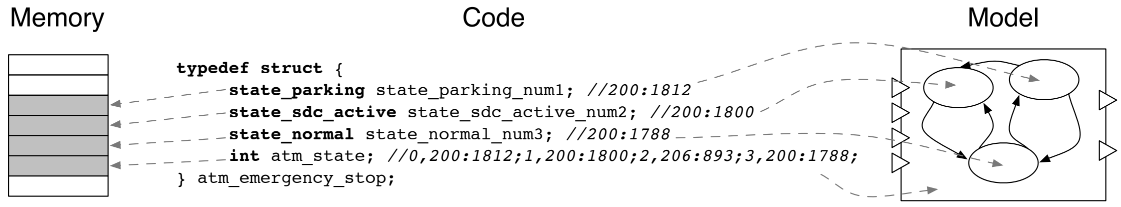


Figure 6.2: Realization of model-level debugging

By using the *code generation* tools, an executable system is derived as described in Chapter 5. To enable model-level debugging, the generated functional code is extended with meta-information about its relation to model elements, as we will describe below. During *system execution*, the actual input and output data of the clusters which shall be debugged are captured. We refer to this activity as *tracing* or *monitoring*. The gathered data are in binary form, and therefore *data extraction* has to take place, before the data can be mapped back to model elements. The resulting *trace* is the model-level representation of the data captured during system execution. Together with the original model, this trace is used to allow for a *replay* at model-level. We will detail the implementation of this concept in the following.

Code generation. Application code is generated for all clusters in the same manner as described in Section 5.4. Only the generation of system configuration files is slightly modified. If a cluster is enabled for tracing in the model editor, it is marked accordingly in the configuration file. The middleware then records a trace for this cluster during execution.

While mapping data captured for ports back to the model is easy using their unique middleware address, the decoding of the clusters' internal states, which are composed of automata states and delays, is more complex. The whole state of a cluster is stored as a struct using a single middleware address. For each sub-unit of a network or state of an automaton, the cluster's struct comprehends a nested struct for keeping the sub-state of the contained model element. To facilitate a mapping of struct members to model elements, the C code is annotated during code generation using unique object identifiers taken from the modeling framework. These annotations are shown as comments in the source code. Since comments are not present in the compiled version of the code, they do not influence system behavior. Hence, the code generator always inserts these comments, no matter if model-level debugging is employed with the target system or not.



```
typedef struct {
    state_parking state_parking_num1; //200:1812
    state_sdc_active state_sdc_active_num2; //200:1800
    state_normal state_normal_num3; //200:1788
    int atm_state; //0,200:1812;1,200:1800;2,206:893;3,200:1788;
} atm_emergency_stop;
```

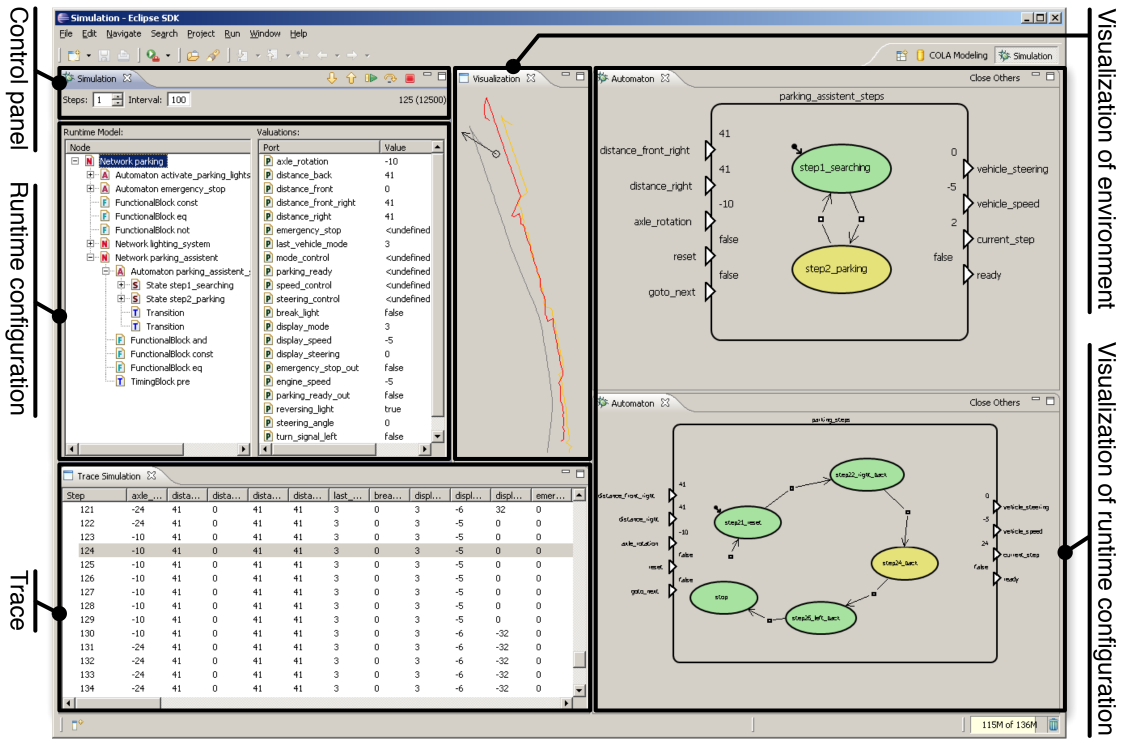
Figure 6.3: Mapping runtime data to model elements

The middle part of Figure 6.3 shows an excerpt of the struct definition for the internal state of a cluster named `atm_emergency_stop`. The top-level unit of the cluster is an automaton. The actual state of the automaton is stored at runtime using an integer variable named `atm_state`. The automaton's states are modeled as sub-units named `state_parking`, `state_sdc_active` and `state_normal`. Each of the sub-units' states is stored in a nested struct. To facilitate a mapping of data back to units in the model, the states' unique identifiers used by the modeling framework are inserted as a comment, next to each variable definition. In the example, these identifiers can be found at each end of line.

System execution. As described before, the middleware handles inter-cluster communication as well as access to the underlying hardware transparently, using numerical identifiers for addressing. In addition, the tasks use the middleware for loading and storing their internal state at the beginning and end of their execution, respectively. Thus, our middleware is able to serve at runtime — besides its primary duty as a mediator for communication — as a monitor for exchanged information and the tasks' internal states.

If tracing is enabled for a cluster, its communication is logged to a file. Since control systems are executed cyclically, this file is expanded during each invocation of the corresponding task. As memory in embedded devices is limited and should not be exhausted, the storage format for the trace has to be as compact as possible. At the same time, no complicated transformations on the data should be carried out to save processing resources. So the proposed storage format is very simple and contains the minimum of necessary information. This includes a header specifying the traced cluster together with its ports and the corresponding addresses as well as the actual data in binary format. In addition, the number of elapsed task activations is stored at the beginning of each invocation, thus allowing to calculate the amount of time which has passed since the start of the system.

Data extraction. For decoding the trace file, it is important to know about the structure of the file. Otherwise, the binary data cannot be transformed into the original values. Therefore, a header is inserted at the beginning of the file. The header is in character representation and defines the values which are stored in the binary part of the file. For each traced cluster a TASK definition specifies the types used for the state values. After the TASK definition the DATA definitions for the ports of the cluster follow, defining port names and data types. If more than one cluster is traced, there may be repeated TASK and DATA definitions. The traced data are saved to the trace file in the same binary representation as they are stored in the memory of the system. Thus, architecture specific properties such as endianness and sizes of primitive data types have to be considered during data extraction. Together with the complex data types specified in the header, it is possible to partition the binary trace into its particular values.



Simulation
Steps: 1 Interval: 100 125 (12500)

Runtime Model:

Node
Network parking
Automaton activate_parking_lights
Automaton emergency_stop
FunctionalBlock const
FunctionalBlock eq
FunctionalBlock not
Network lighting_system
Network parking_assistent
Automaton parking_assistent_s
State step1_searching
State step2_parking
Transition
Transition
FunctionalBlock and
FunctionalBlock const
FunctionalBlock eq
TimingBlock pre

Valuations:

Port	Value
axle_rotation	-10
distance_back	41
distance_front	0
distance_front_right	41
distance_right	41
emergency_stop	<undefined
last_vehicle_mode	3
mode_control	<undefined
parking_ready	<undefined
speed_control	<undefined
steering_control	<undefined
break_light	false
display_mode	3
display_speed	-5
display_steering	0
emergency_stop_out	false
engine_speed	-5
parking_ready_out	false
reversing_light	true
steering_angle	0
turn_signal_left	false

Trace Simulation

Step	axle_...	dista...	dista...	dista...	dista...	last_...	brea...	displ...	displ...	displ...	emer...
121	-24	41	0	41	41	3	0	3	-6	32	0
122	-24	41	0	41	41	3	0	3	-5	0	0
123	-10	41	0	41	41	3	0	3	-5	0	0
124	-10	41	0	41	41	3	0	3	-5	0	0
125	-10	41	0	41	41	3	0	3	-5	0	0
126	-10	41	0	41	41	3	0	3	-5	0	0
127	-10	41	0	41	41	3	0	3	-5	0	0
128	-10	41	0	41	41	3	0	3	-5	0	0
129	-10	41	0	41	41	3	0	3	-5	0	0
130	-10	41	0	41	41	3	0	3	-6	-32	0
131	-24	41	0	41	41	3	0	3	-6	-32	0
132	-24	41	0	41	41	3	0	3	-6	-32	0
133	-24	41	0	41	41	3	0	3	-6	-32	0
134	-24	41	0	41	41	3	0	3	-6	-32	0

Figure 6.4: User interface of the simulator

For mapping port values from the trace to elements in the model, the corresponding ports can be found via cluster and port names specified in the header. Mapping state values contained in a struct to model elements is more difficult, since the corresponding elements are distributed along different refinement layers in the model. The values for all these elements are combined in one binary chunk extracted from the trace, namely the state of the cluster. In order to partition the state struct into its members, the application's source code is considered. Using the struct definition, the binary data can be divided into struct elements. To identify the model unit associated with a struct element, the numerical identifier of the unit is looked up in the annotated C code. In Figure 6.3, these identifiers can be seen in the form of comments in the C code. Figure 6.3 also indicates the mapping of actual runtime data in the system to model elements using their identifiers.

Two basic types of state information can occur in the trace: current values of delays or the current state of an automaton. For delays the stored values simply have to be mapped to the model element. In contrast, an automaton cannot be just provided with some value, but has to be mapped to the unit implementing the currently active state. In the code the active state is identified by an integer value. The struct member storing the active state is annotated with mappings from the state number to the identifier of the unit implementing the state. With this information, the corresponding unit can be set as the active state in the automaton during simulation.

As can be seen in Figure 6.3, nested structs are annotated with one identifier, while the integer variable `atm_state` is annotated with a list of identifiers separated by semicolons. This list is necessary because the `atm_state` holds the active state of the automaton encoded as an integer value in the C code. Each token of the list contains the integer representation of an automaton state and the unique identifier of the corresponding automaton state in the model. Using this information, the integer value found in the trace can be mapped to an automaton state in the model.

After data extraction is finished, the recorded trace can be loaded into the simulator to replay the execution at model-level. The meta-model of a trace is clearly defined and is part of the integrated modeling language COLA, which has been presented in Section 4.3. Conceptually, the trace consists of a sequence of execution *steps*. For each execution step, it logs enough information to be able to repeat the execution in the simulator. First, the *valuations* of the ports at the cluster's boundary are stored. This also includes sources and sinks that are contained in the cluster. Whereas the valuations of the input ports are required to retrace the execution, the valuations of the output ports are used to validate the executed code against the model-level simulator. Second, the *states* of all stateful units that are part of the cluster are saved. Whereas the state during the first step is required to initialize the execution in the simulator, the states of the following steps are also used for validation.

The actual syntax defined by our modeling language represents traces by means of two-dimensional tables. Figure 6.4 depicts the simulator with an example trace loaded. The according table has a row for each execution step, and a column for each port at the cluster's boundary as well as for each stateful unit contained in the cluster.

Replaying traces. We extended the existing simulator [71] with the functionality to replay a trace recorded on the hardware platform. The simulator then executes the units which are included in the traced cluster. Due to the modular architecture of our simulator, the realization of this functionality could be integrated in a straight forward manner. The architecture of the simulator is modularized into three components: the runtime configuration, the environment interface and the execution strategy.

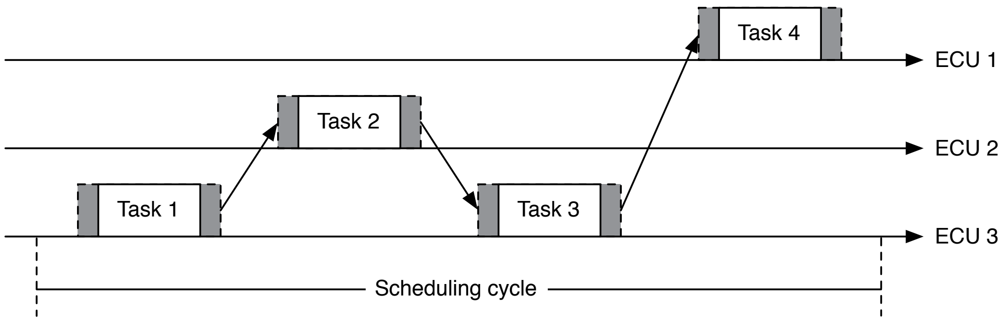
The *runtime configuration* describes the state of a cluster during runtime. That means, it equips the clustered units with information required at runtime. These include the valuations of ports and the internal states of stateful units. The initial runtime model can be automatically derived from the structure of the clustered units. The *environment interface* specifies the behavior of the environment surrounding the cluster. Therefore, it controls the inputs entering and observes the outputs exiting the ports at the cluster's boundary. Based on the environment interface provided by the simulator, different kinds of environments can be realized. The replay of a trace is implemented by constructing an environment interface from the trace that provides the simulator with the monitored information. The *execution strategy* performs a stepwise execution of the system. This is done by modifying the runtime configuration depending on the inputs and generating the outputs of the cluster. The execution strategy is determined by the semantics of our modeling language.

Figure 6.4 shows the user interface of our simulator during the replay of a trace. The trace was recorded while testing the functionality of a case study on the hardware platform. A control panel allows the user to start, pause and stop simulation as well as to regulate its speed. The user interface provides views for both the structure and the visualization of the runtime configuration in real syntax. An optional visualization shows the target system's environment.

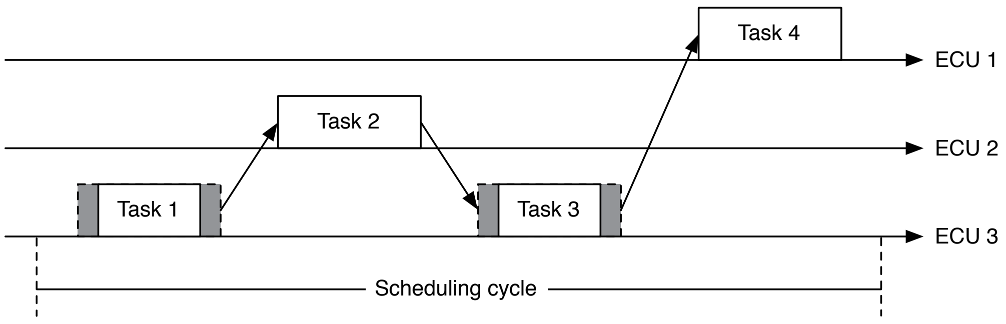
6.1.5 Influence on the Target System

It is well known that any kind of monitoring system, like our tracing approach, is at risk to change the behavior of the monitored system. McDowell and Helmbold discuss this problem in their survey about debugging techniques for concurrent programs [100]. The effect has been referred to as the *Heisenberg uncertainty principle* in recording traces for debugging by LeDoux and Parker in [87] or as *probe effect* by Gait in [49]. For concurrent systems the change in system behavior arises from the fact that the monitoring code itself consumes resources like memory and processing power, and thus the execution of the application programs is delayed or hindered. Therefore race conditions which were present in the original and unmonitored system may be altered. Furthermore, the additional memory consumption for the monitoring code and the gathered traces might exceed the amount of memory available in the target system.

If, however, a system is scheduled statically according to a global clock, influence regarding race conditions could be avoided. This requires the scheduler to consider all possible delays introduced by the monitoring. Hence, the scheduler adds the



(a) Tracing disabled



(b) Tracing enabled for `T2` and `T4`

Figure 6.5: Comparison of schedules with and without tracing

time needed for storing input and output data to the WCET of every task, no matter whether the tracing is executed for the respective task or not. The result is a schedule which includes an amount of slack time that is equal to the amount of time needed for tracing all tasks.

Figure 6.5 gives a comparison of two alternative schedules for a system of three ECUs. The schedule in subfigure (a) depicts the regular schedule without any tracing enabled. There is some slack time before and after each task, indicated in gray in the picture, which is reserved for tracing. In subfigure (b) the same schedule is shown, except that tracing has been enabled for `Task 2` and `Task 4`. The slack time surrounding these tasks is now used for tracing. Hence, the runtime of `Task 2` and `Task 4` is extended by the length of the slack time. As can be seen in Figure 6.5, the starting times for all tasks are identical, no matter whether tracing is enabled or not. There is no change in system behavior because of a influence on the system's timing. Using timestamps which are stored together with the data in the trace file, it is possible to match the respective data of tasks executed on different ECUs with each other.

From the above description it should be clear that the reservation of slack time, which may be used for tracing, lengthens the execution cycle of the system schedule. Thus the concept of providing slack time for tracing in every schedule might not be applicable if this results in a violation of deadlines. However, omitting the slack time in the schedule results in a system with different timing behavior compared to one with the tracing function enabled. Since the COLA deployment approach generates a static system with fixed causal ordering of tasks, one might think that the overall system behavior should still be identical. Nevertheless, since we are dealing with a control loop system, the changed duration between input and output — which might be within the specified deadline in both cases — may result in a different behavior due to the altered feedback time between producing output and subsequently reading possibly related input.

If in contrast the system provides enough computing resources to allow the inclusion of slack time in the schedule, the timing of the generated system will be identical, no matter if tracing is enabled or not. The impact that prevails is memory consumption. There is a need for working memory to execute the tracing function and to store the monitored data. While the overhead for the tracing program code is very small, the amount of memory for a trace can grow considerably, depending on the size of the monitored data and the duration of the monitoring. As the memory resources of embedded systems are in general very limited, there might be very little free memory left which therefore limits the possible size of a trace. As a consequence it might be necessary to add some additional memory to the hardware platform which is used for debugging.

Concluding we can say that the described debugging approach can be implemented with minimal and well-known impact onto the target system.

6.2 Generation of Fault Tolerance Modes

In order to ensure the safe operation of a real-time system, its software as well as its hardware have to be functional. Using the afore described concepts for programming such systems, the quality of software may well be improved. Still, systems resulting from a MDD process bear the risk to fail because of hardware defects. Any approach avoiding such failures is highly welcome, especially in safety-critical system like cars or airplanes. While the avionic industry tackles the problem using redundant software and hardware, automotive systems suffer from a much higher cost pressure. The large number of units produced prohibits the employment of spare hardware or concepts like 2-out-of-3 systems due to their huge cost and energy overhead. Further, the additional weight and necessary installation space would be a huge problem in an automobile.

However, employing appropriate allocation and scheduling mechanisms would allow the use of spare processing capacity on nodes already present in the distributed computing system of a car. This spare capacity, i. e., memory and processing cycles, does exist in most systems since it is generally not possible to select a platform which exactly meets the requirements. Instead the most cost-efficient available on the market is selected, providing at least the needed resources. Relocating the most important tasks from one ECU to another in case of a hardware failure would thus be feasible, if the overall system is not endangered by the transition into the fault tolerance mode.

The approach presented in this thesis does neither employ any new failure detection method nor is it able to deal with all possible failures. Instead its contribution lies in its integration into our deployment approach, which disburdens the developer from programming error handling routines. This way the developer may focus during his system design on the intended functionality, while neglecting a major amount of effort for error handling. The deployment system is able to generate the fault tolerance code on its own.

Support for the unattended integration of fault tolerance mechanisms into safety-critical systems is very desirable. As a study by Mackall and colleagues showed [94], all failures observed in the reviewed system where due to bugs in the design of the fault tolerance mechanisms themselves. And as Rushby showed in his overview of fault-tolerant bus architectures [115], even up to date bus systems like FlexRay do not necessarily have integrated fault tolerance mechanisms. In contrast, the concept presented here yields reliable fault tolerance modes by automatically generating the appropriate code from the COLA model. Further, the concept is not dependent on hardware support by the bus system. Instead it can be integrated into our middleware, as we will show.

6.2.1 Fault Hypothesis

The generated fault tolerance modes are intended to guarantee the continuous operation of highly safety-critical tasks in case of a hardware failure. The presented approach makes use of our concept for operating modes described in Section 5.6. As explained, this operating mode concept implements a switching of schedules at runtime. The respective modes are pre-calculated offline which guarantees deterministic system behavior. The schedules differ regarding which tasks are activated and their respective starting times. By relying on the time-triggered paradigm, a synchronous change of schedules at runtime can be ensured.

As the spare capacity in an automotive system which we intend to make use of is very limited, it is impossible to compensate an arbitrary number of ECU failures. Rather we intend to guarantee the substituted execution of few most important tasks from a single failing ECU on other ECUs containing spare resources. The decision about which tasks are elected for redundancy is up to the developer and shall be specified in the software model. Using this information in combination with the hardware platform model, a suitable fail-over ECU is calculated for each task. The necessary allocation scheme as well as the changed schedule may then be determined by our deployment tools and selected by our middleware.

The concept outlined here is able to deal with the complete loss of a single

ECU. While this case may seem synthetic, it is quite common in practice. In an automotive system the ECUs as well as their connecting cables are exposed to permanent vibrations and changing environmental conditions like temperature, humidity, etc. These conditions can easily lead to broken wires, either in form of copper wires or the even more sensitive fiber optic cables which may be used for example with the FlexRay or MOST bus systems. Other defects might be loose contacts or cracks in circuit boards. Thus, while the described approach does not address problems like a “babbling idiot” at the moment, a safe operating state can be guaranteed for the typical case of a not responding ECU, may it be because of communication loss, power loss, or loose electronic components.

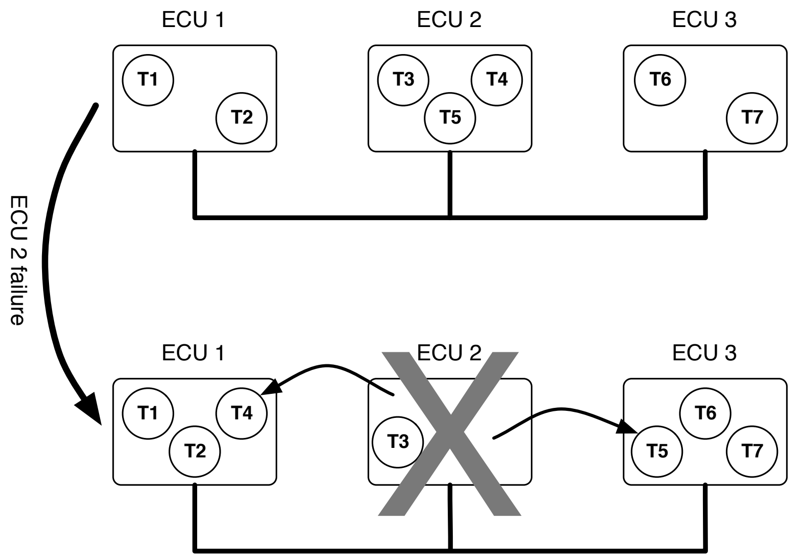


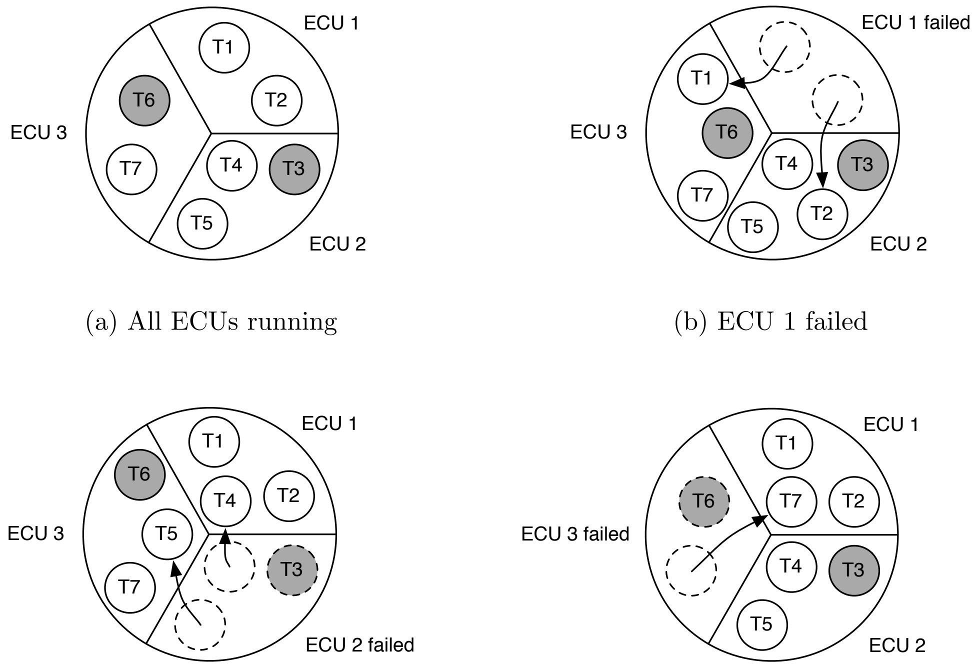
Figure 6.6: Task transition in case of failure

Figure 6.6 shows an example for a switch to a fault tolerance mode. The upper half of the figure shows three ECUs and their respective executed tasks `T1` through `T7`. In the lower half a possible fault tolerance mode is shown, which is activated by the failure of `ECU 2`. In the exemplary fault tolerance mode the tasks `T4` and `T5` are necessary for the safe operation of the system. Thus they are executed on `ECU 1` and `ECU 3` respectively, if `ECU 2` fails. Task `T3` is not marked safety-critical in the model and thus it is ignored in the fault tolerance mode.

The concept requires sensors and actuators being used by safety-critical tasks to be directly connected to the bus or to have a redundant connection to the failover ECU. Otherwise they would not be accessible in case the ECU they are connected to fails and thus the according task could not continue its operation successfully.

6.2.2 Redundant Allocation

The original allocation mechanism, described in the deployment chapter, tries to find one valid allocation for the clusters given in a COLA model to the respective hardware model. Since the fault tolerance concept is aimed at masking the loss of an ECU, the hardware model has to be modified accordingly. Instead of having exactly one hardware model for the platform, the fault-tolerant deployment considers $n+1$ hardware models for a platform consisting of n nodes. One model represents the original platform with all nodes operational, and the n other models correspond alternatively to the platform with one of its ECUs failing.



(a) All ECUs running

(b) ECU 1 failed

(c) ECU 2 failed

(d) ECU 3 failed

Figure 6.7: Possible fault tolerance modes

Figure 6.7 shows an example of the substitute allocation schemes calculated for the platform from Figure 6.6, which consists of three ECUs. First, in subfigure (a) an allocation for the fully functional platform is given. In subfigures (b) through (d) one of the ECUs failed in each case. Hence, all tasks which are allocated to it and have been indicated as safety-critical by the developer — in the example all tasks except the grey marked tasks `T3` and `T6` — are migrated to the remaining ECUs. Their migration is indicated by small arrows. At deployment time, the allocation routine considers each of these cases and calculates a possible allocation for it.

When a solution has been found, the according schedule is constructed. If the allocation is not schedulable, a different allocation has to be calculated.

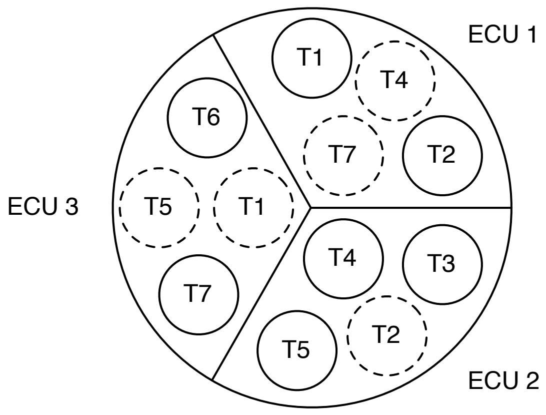


Figure 6.8: Overview of all tasks and their redundant allocation

To allow the execution of a cluster on several ECUs requires the code generated for it to be placed onto all ECUs in question, according to the derived allocations. With the code, as well as the different schedules available, the restart of a task on a different ECU is facilitated. Figure 6.8 shows the overall allocation of all tasks, providing redundancy for the failure of any of the ECUs. The original allocated tasks are indicated by a solid boundary line, while their redundancy is drawn with a dashed circle. For the non safety-critical tasks `T3` and `T6` no redundancy is provided.

To detect the need for such a change in schedules, a means of failure detection is needed, which we will explain next.

6.2.3 Hardware Failure Detection

In order to deal with hardware failures, a means of failure detection is necessary. As described before, we aim at addressing defects causing a loss of communication to a node of the system. We intend to detect a failing ECU externally by the other ECUs of the system, as this allows for failure detection even in case of a broken bus connection. The use of additional hardware for failure detection would lead to additional cost and is not acceptable for an automotive system. Considering the time-triggered nature of the intended target system, two approaches for failure detection come to mind. The solutions differ in that one is passive and the other is active. Both concepts may be integrated into our communication middleware.

First, it would be possible to detect the omission of an anticipated message. Since all communication points are known in advance, the omission of a message indicates a fault in the respective sending node. This assumption is only valid for the observation of the overall system, if every node sends at least one message. But in theory, there might be nodes which only receive messages, so their defect might

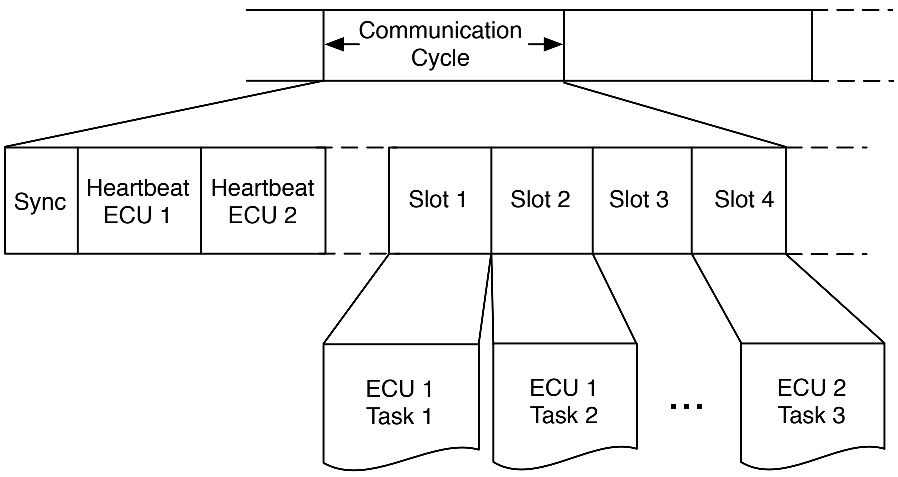


Figure 6.9: Heartbeats in the communication cycle

remain unnoticed. This first approach is passive since it does not require any action by the nodes besides their regular operation.

Second, each node in the system could send a cyclic heartbeat message to indicate being operational. Consequently a failed ECU is identified by the other ECUs in the system, if the heartbeat was not received within a defined window. Because of the need for sending heartbeat messages this approach can be considered active.

For our fault tolerance approach we prefer the active approach, because of the possibility to identify any faulty nodes. The implementation of the active approach requires every node in the system to send a heartbeat, which can ideally be done using a broadcast at the beginning of every communication cycle. More precisely, first the network master ECU sends its clock synchronization to the slaves, as indicated in Figure 6.9. Following this message each node has an assigned slot for sending its heartbeat message. Every ECU in the system keeps a status vector containing the timestamp of the last heartbeat of all nodes in the system. This vector can then be used to detect faulty nodes. To avoid a split decision when to change into the fault tolerance mode, the network master is responsible for deciding and communicating the mode change to the slaves. An arbitrary threshold may be specified for the number of missing heartbeat messages before an ECU is identified as faulty. This decision might depend on the system in question. Updating the status vector in every node of the system and not just the network master enables all slaves to inherit the master status if the original master node fails. Further, a node which observes continuously increasing delays for all values in the vector might consider itself as disconnected from the network and may shut down or change into a fail-safe mode which could mean sending safe default values to all connected actuators.

6.2.4 Switching to Fault Tolerance Modes

As presented in Section 5.6, a middleware address can also be used to communicate operating modes, which triggers a synchronous change of dispatching plans on all nodes of the system. This ability is especially of importance for the fault tolerance concept presented here. A fault tolerance mode is essentially an additional operating mode, which defines the execution of safety-critical tasks on a different ECU, in case of failure of the originally chosen ECU.

As mentioned before, a system schedule is valid for a certain allocation. For each additional allocation another schedule is calculated. Therefore the concept introduced here results in several scheduling graphs, each of which is valid for either the default allocation or one fault-tolerant allocation. At the beginning of each cyclic execution, the middleware checks on system health by exchanging heartbeat messages. The network master is then able to communicate the actual mode, which is either `default` or `ECU X failed`, using a fixed middleware address. Subsequently the dispatchers of all ECUs select the appropriate task set to execute. The resulting system corresponds to the example in Figure 6.6, where `ECU2` failed and the dispatchers of `ECU1` and `ECU3` change their activated schedule in order to migrate the safety-critical tasks `T4` and `T5`.

In principle the middleware also supports the migration of the actual task state. For this purpose cluster states may be replicated to other ECUs in each communication cycle. Whether this migration is desirable depends on the cluster in question as the threshold of omitted messages employed for failure detection might render the state outdated.

In summary the described concept is suited to increase the reliability of the generated system considerably. Just by specifying safety-relevant clusters in the model, a system engineer is able to trigger the creation of a system being able to continue its safe operation even in presence of a hardware defect. This state in which the system is not fully functional anymore, but provides safe operation is often referred to as *graceful degradation*, consider Gärtner's work about fault-tolerant distributed computing [50].

6.3 Adaption to AUTOSAR

As described in Section 3.3, the automotive industry is currently adapting the AUTOSAR standard for modeling system architecture. With AUTOSAR becoming the de-facto standard for automotive platforms, a concept for integrating COLA designs into an AUTOSAR development process is required. Such a concept would allow a stepwise migration from AUTOSAR to COLA. Otherwise, the utilization of COLA would require a complete redevelopment of the existing system, which results in an excessive increase of development time and cost and is, therefore, not feasible.

Since both the COLA and the AUTOSAR approach are based on custom middleware concepts, the question arises which middleware to use for a hybrid system. The answer can be given by considering the way software is implemented in the two concepts. Using the afore described COLA deployment approach would require the entire system to be modeled in COLA. During code generation, the modeled functionality is transformed into code which interfaces the middleware described in Section 5.2.3 at runtime. This middleware features a fixed interface and the COLA clusters are transformed into code which matches this interface. In an AUTOSAR development, on the contrary, the RTE's interface is generated according to the respective interfaces of the connected SWCs. Thus, AUTOSAR SWCs cannot be executed on the COLA middleware without conceptual changes of either the middleware, or the AUTOSAR SWCs due to incompatible interfaces. It is however feasible to generate an AUTOSAR RTE specification from a COLA model. An RTE being generated based on this description enables the integration of code generated from COLA clusters into an AUTOSAR system. Hence, as a hybrid scenario, we will describe the integration of COLA clusters into an AUTOSAR system, and not the use of AUTOSAR SWCs in a COLA based system.

Because of the differing modeling and platform concepts, the execution of COLA clusters on an AUTOSAR platform yields several kinds of problems. In the following sections we will summarize the characteristics of the COLA and AUTOSAR platforms. Subsequently, we will explain the arising problem areas. Finally, possible solutions for the problem areas and a possible integration process are presented.

6.3.1 COLA Target Platform

The details of the COLA target platform have been presented in Section 5.2. In short the platform consists of operating system and middleware. The operating system facilitates the non-preemptive execution of tasks to a given static schedule. The middleware provides a global time to all nodes of the distributed system, which is used to synchronize task execution on the respective nodes. Further, communication between tasks in the system is only allowed in predefined time slots, which are related to the global clock, too. The middleware is also employed to access sensors and actuators. According to the employed time-triggered concept, the middleware cyclically polls sensors and buffers the received values. These data may subsequently be accessed by the tasks of the system. Configuration of the middleware is generated from the COLA model and stored in a configuration file. This file is read by the actual middleware instances at system startup and the instances are configured accordingly.

6.3.2 AUTOSAR Target Platform

An AUTOSAR target platform consists of an AUTOSAR OS and the AUTOSAR RTE. The AUTOSAR OS specification is not limited to a specific operating system, though OSEK is a common choice for most systems [9]. The AUTOSAR RTE cannot be configured in the final system, but is intended to be a static middleware layer. AUTOSAR does, however, facilitate the generation of code for the AUTOSAR RTE according to the respective architecture model. Thus, the RTE is configured offline and specifically for each system.

A cyclic execution of tasks is possible according to the AUTOSAR OS standard. However these tasks are executed preemptively and, hence, may be interrupted by another higher-priority task. Another source for task preemption are external interrupts which trigger execution of OS kernel code like device drivers. Thus, a strict time-triggered execution of tasks is not possible. Besides cyclic tasks, the AUTOSAR OS allows the event-triggered execution of tasks. Since the respective interrupts might occur at unknown points in time, the timing behavior of such a system cannot be predicted in the general case. Rather, assumptions about the minimum interarrival time of interrupts and spare hardware resources have to be employed to comply with the required real-time requirements.

In contrast to the COLA middleware, the RTE is generated specifically for every ECU according to the respective AUTOSAR model. Its interface is fixed and may not be configured in the final system. As a result, the RTE produces very little memory and processing overhead. But this approach also requires a replacement of all RTE instances in a system, if the interface of a SWC is changed or another SWC is added after the RTE has been generated.

The timing behavior of communication via the RTE is based on the bus systems currently employed in automotive systems. Synchronization of clocks between ECUs is not offered by the RTE due to a lack of this function from several of the employed bus protocols, like CAN and LIN. AUTOSAR proposes two event-based communication paradigms, namely client-server and sender-receiver communication. When sender-receiver communication is used, the sender transmits messages asynchronously. It may continue executing its tasks immediately after calling the sending routine. The receiver is notified by an interrupt, when messages have been received. Using client-server communication the function call for sending a message blocks the client until a response is received from the server. The arrival of a message, again, causes an interrupt at the receiver side.

In contrast to AUTOSAR, every exchange of data happens at predefined points in time in a COLA-based system. The according system-wide schedule is calculated offline and ensures the time-triggered execution of the system within its required deadlines. This renders the use of interrupts during message exchange redundant.

6.3.3 Integration Concept

In order to facilitate an integration of code generated from a COLA cluster into an AUTOSAR system, compatibility of the interfaces of the cluster and of the RTE which executes the cluster have to be established. To this end, the cluster has to be replaced by an appropriate AUTOSAR counterpart. Comparing the modeling artifacts of the two languages, COLA clusters are closest to AUTOSAR SWCs. An AUTOSAR SWC is a self-contained piece of software which has its own private memory and interacts with its environment by means of ports. These ports are either of sender-receiver or client-server type, which implies the according communication paradigms described before. Just like clusters, SWCs may only be allocated to an ECU of the target platform as a whole. Each SWC may define a number of *runnables* which are implemented by functions that may be scheduled by the operating system.

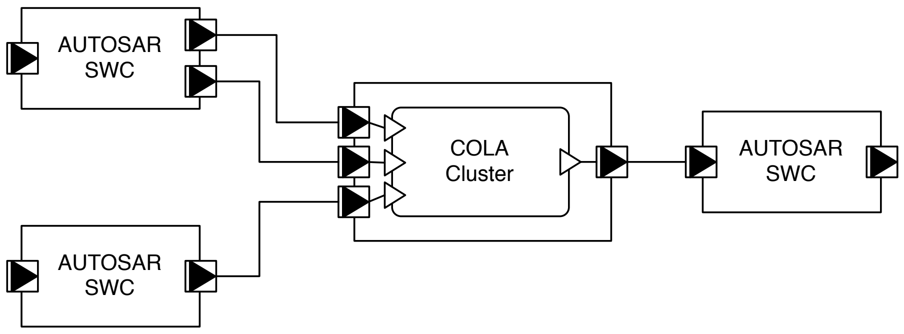


Figure 6.10: COLA cluster wrapped in an AUTOSAR SWC

As indicated in Figure 6.10, the concept for integrating a COLA cluster into an AUTOSAR system is based on wrapping the cluster into an AUTOSAR SWC. By equipping the cluster with an AUTOSAR interface, the code for that cluster becomes AUTOSAR compatible. This may be achieved by using a modified version of the code generator described in Section 5.4, which provides the cluster with the interface of a SWC. But this code generation is limited to the cluster itself. The integration into the AUTOSAR platform by means of a matching RTE interface is still needed.

Regarding the regular AUTOSAR workflow, the modeled architecture is used to generate the RTEs for all nodes of the target platform. As a result, the SWCs can be linked to these RTE instances. When integrating a cluster from a COLA model into an AUTOSAR system, this cluster is not present within the AUTOSAR model. Consequently, the RTE generator would not produce code for interfacing that cluster. Therefore, the cluster has to be declared to the RTE generator before the RTE code is produced. This can be achieved by inserting the appropriate data into the AUTOSAR model before passing the designed architecture to the RTE

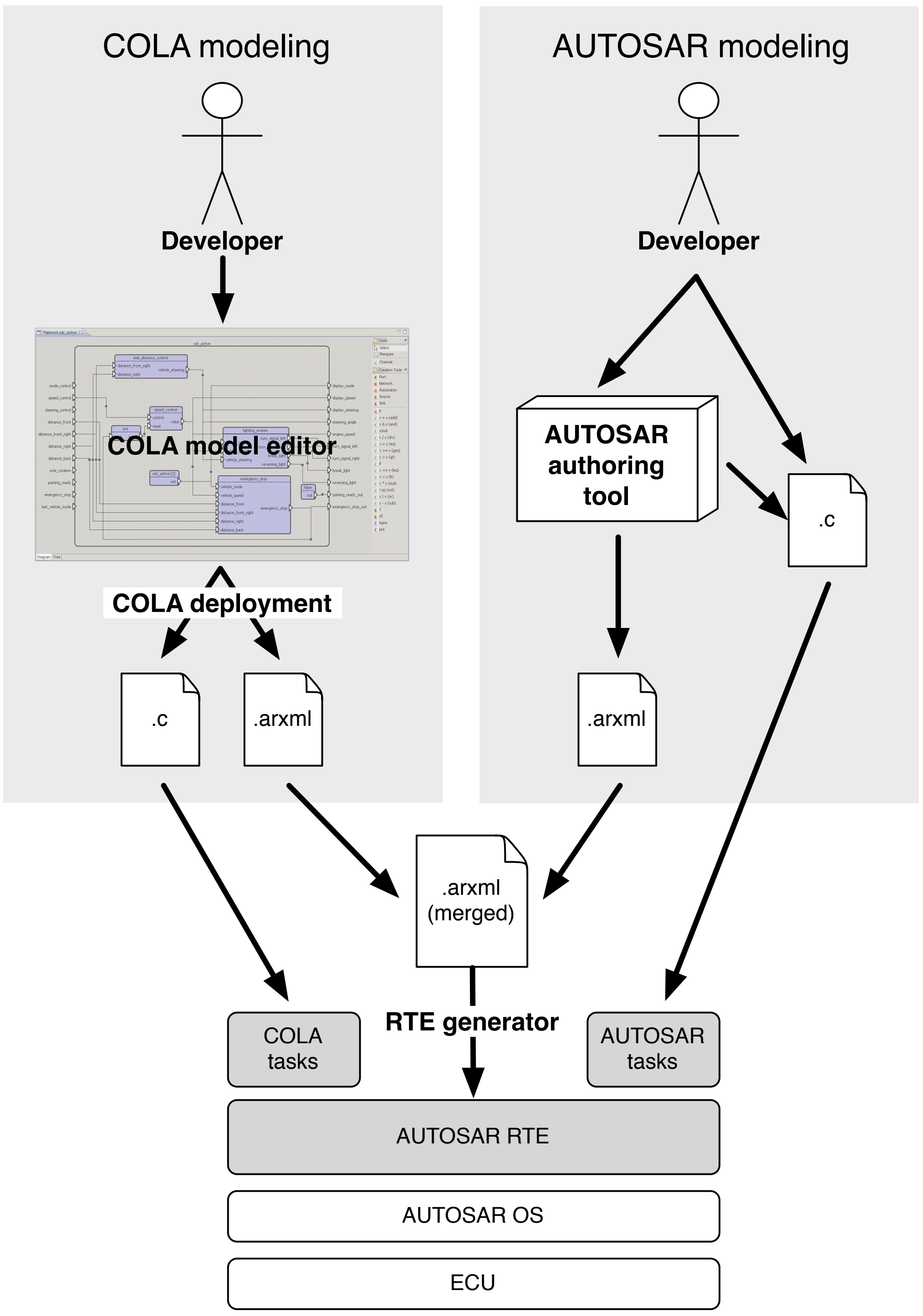


Figure 6.11: Workflow for integration of COLA and AUTOSAR

generator. AUTOSAR defines the AUTOSAR XML format as a data format for exchanging designs between tools used throughout the development process. Data may also be distributed over several of the so-called `.arxml` files. Thus, if an appropriate `.arxml` file is available for the COLA cluster, it may be used together with the respective file of the remaining AUTOSAR design as input for the RTE generator.

Figure 6.11 gives an overview of the described concept. On the left side of the figure, modeling of parts of the system is carried out using the COLA editor. The resulting clusters are generated into C code comprising the clusters' behavior. Further an AUTOSAR XML description which defines the interfaces of the clusters is generated. The right side of the figure depicts modeling using an AUTOSAR CASE-tool. The CASE-tool outputs C code templates which the developers use as a basis for implementing the desired behavior. Further, the CASE-tool produces an AUTOSAR XML file comprising the designed architecture. The two `.arxml` files are merged and used by the RTE generator to produce appropriate RTEs for the platform. As a result, the application code generated from COLA can be executed alongside the applications designed in AUTOSAR on the target platform.

Considering the afore described workflow a technique for modeling communication between AUTOSAR SWCs and COLA clusters is missing. For the communication of COLA clusters executed on an AUTOSAR platform we intend to use the AUTOSAR RTE instead for the COLA middleware. To facilitate this communication, the clusters modeled in COLA have to be also present in the AUTOSAR model. This allows the SWCs from the AUTOSAR model to interface them. In addition, the RTE code generator produces an RTE instance which provides an interface to the COLA cluster.

To achieve this integration, each COLA cluster has to be represented by an equivalent SWC in the AUTOSAR model. This dummy SWC must feature the same interface as the COLA cluster, regarding its ports and runnables. This enables the SWC to be connected to other SWCs in the AUTOSAR model. Later, during combination of the two AUTOSAR XML files, this dummy SWC has to be replaced by the XML specification generated from the COLA model.

The described integration concept may only work, if changes are made to the COLA code generator. Further, special care has to be employed when interfacing the COLA cluster from an AUTOSAR SWC. We will describe the necessary changes in the following section.

6.3.4 Necessary Changes

AUTOSAR and COLA modeling yield several conceptual differences. Due to these differences, hybrid modeling using both approaches requires a concept for mapping elements of one language to equivalent elements of the other language. This is especially crucial at the interfaces where elements from both languages interact. In addition, the execution semantics of both languages are different regarding their scheduling. We will outline each of the crucial problem areas in the following and propose a solution where possible.

Communication. The COLA deployment tools insert numerical addresses as a parameter into the middleware calls used for communication. Using these addresses the middleware is able to transmit the message to the respective communication partner. The signatures of the according middleware functions have been presented in Listing 5.1.

The AUTOSAR RTE, in contrast, provides a particular function for every port featured by a SWC. If a COLA cluster shall be executed on an AUTOSAR platform, it has to use these functions instead of the COLA middleware API. Listing 6.1 shows the signature of the RTE communication functions defined in the RTE documentation [10]. In the function names, `<p>` is replaced by the respective port name and `<o>` specifies the data object, that is the data type of the port.

```
Std_ReturnType Rte_Write_<p>_<o>(IN <data>)
Std_ReturnType Rte_Read_<p>_<o>(OUT <data>)
```

Listing 6.1: Interfaces for sender-receiver communication

In AUTOSAR distinct ports are used for sending and receiving messages, when using the sender-receiver concept. For the client-server paradigm, a single port is used for request and response. Since the concept of COLA ports does not allow bidirectional communication, only sender-receiver ports should be used in the AUTOSAR model when interfacing COLA clusters.

Regarding the identified issues, the integration of a COLA cluster into an AUTOSAR system requires the COLA code generator to produce AUTOSAR compatible send and receive functions according to the signatures in Listing 6.1. It should be clear that the port names defined in the AUTOSAR model have to be available in the COLA model for generating appropriate communication calls to the RTE. Thus, ports should have the same names in COLA, as their counterparts used for the dummy SWC in the AUTOSAR model. This enables the code generator to produce compatible communication calls. Further, only sender-receiver ports may be used at the interface between COLA and AUTOSAR components.

Interfacing Devices. Using the COLA approach, access to sensors and actuators is carried out by the middleware. The device drivers are interfaced by the middleware, exclusively. The code generated from a COLA cluster may interface the devices using the according middleware functions, which require the numerical address of the device, as well as a pointer to the memory region data shall be read from or written to, and the length of data to read or write. The signatures of the respective functions have been presented in Listing 5.1.

In AUTOSAR, on the contrary, access to hardware drivers is realized by the RTE and may use arbitrary function names. The number of parameters and their types are not fixed, as well. Furthermore, the RTE serves solely as a mediator which forwards the function calls executed by the applications. For most hardware devices hardware access is performed immediately according to the client-server paradigm. There is no polling concept available in the RTE, as is the case with COLA. Hence, hardware interfacing hardware is realized in a synchronous manner, i. e., the application is blocked until the device driver returns from execution.

As described before, COLA does not support client-server communication. It is therefore not possible to interface an AUTOSAR device driver directly. However, the software layering concept of AUTOSAR envisions the use of sensor and actuator SWCs, as outlined in the description of the AUTOSAR architecture in Section 3.3. These SWCs are used to provide a common interface for a certain type of device, no matter which manufacturer built the respective device. Thus, applications may use the device unaltered, even if the manufacturer differs across different systems. For accessing devices from code generated for a COLA cluster, these sensor and actuator SWCs could be modified to provide sender-receiver interfaces. This way they could be used by COLA clusters.

Storage of Internal Task States. In an embedded control system tasks often have an internal state which is retained between subsequent invocations. Tasks generated from the COLA approach employ the middleware for storing their internal between execution cycles. When executing such tasks in an AUTOSAR system, a suitable alternative has to be used.

According to the AUTOSAR standard, each SWC may define a private memory region, called *per instance memory* (PIM). The size of this memory region is defined in the `.arxml` file of the respective SWC. A handle to this PIM may be requested by calling the function shown in Listing 6.2 [10]. The PIM referenced is of the data type `<type>` and can be identified by its given `<name>`.

```
<type> Rte_Pim_<name>()
```

Listing 6.2: Call for accessing PIM

During generation of code for a COLA cluster, the modified code generator has to insert calls to this PIM for accessing the task's internal state. Further, the PIM has to be defined in the respective `.arxml` file.

6.3.5 Timing Behavior of the Hybrid System

The deployment approach described in Chapter 5 transforms a COLA model into a time-triggered system. The schedule which defines starting times for tasks as well as communication slots for the system is calculated offline. For the timely execution of this schedule a global time source is needed. Systems which use the COLA deployment approach are provided with a global clock by the middleware. This clock is synchronized regularly to provide the necessary accuracy on all nodes of the system.

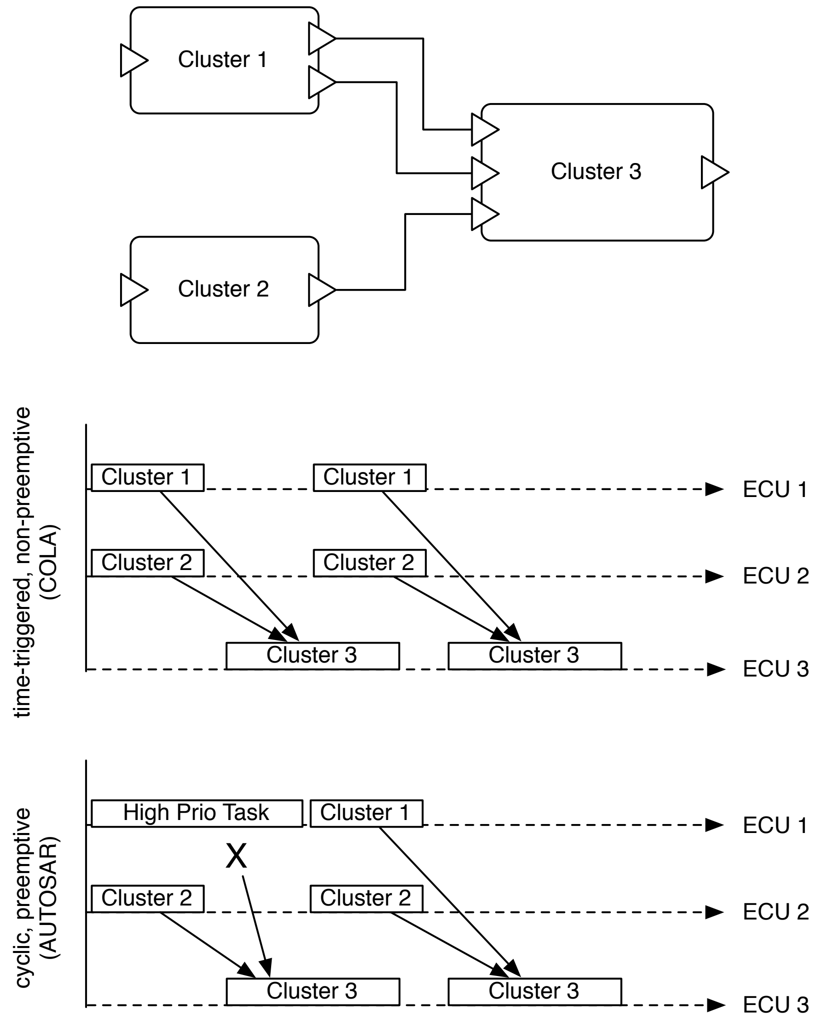


Figure 6.12: Comparison of COLA and AUTOSAR scheduling

AUTOSAR systems, however, operate on a event-triggered basis [11]. Execution of tasks as well as communication is triggered by external events, by other tasks, or by interrupts generated by a timer. Without the need for a global time, there is no global clock or synchronization provided.

The different execution semantics of COLA and AUTOSAR result in differing scheduling schemes. In a system created by the COLA deployment tools, all tasks are executed non-preemptively according to the pre-calculated time-triggered schedule. AUTOSAR in contrast uses a priority based, preemptive scheduling approach with priority ceiling [12]. Thus, if COLA clusters are executed in an AUTOSAR system, the assumptions made about their execution order may be incorrect. It is possible to define the COLA clusters to be executed as cyclic tasks in the AUTOSAR system. This resembles best the synchronous timing semantics. However, cyclic execution on an AUTOSAR platform may yield inaccurate timing due to preemption.

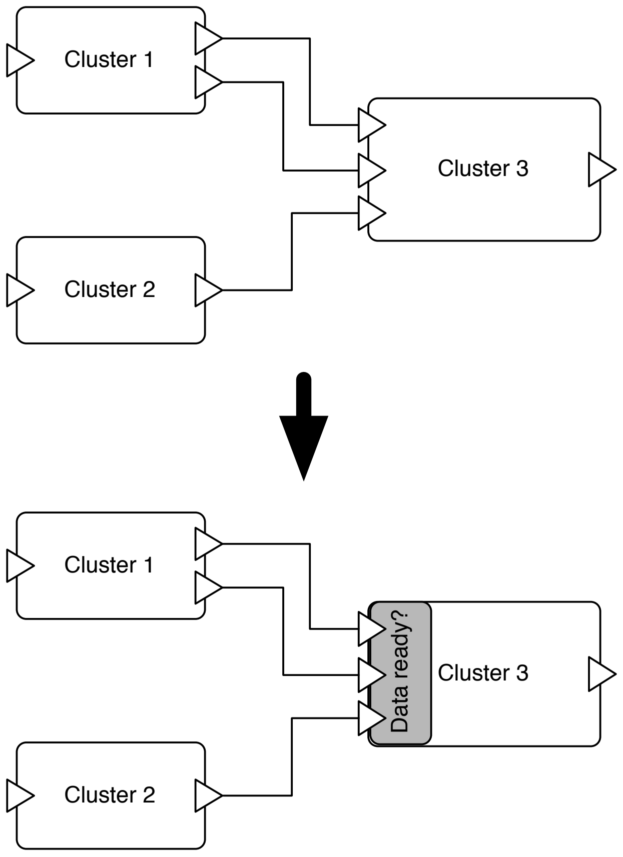


Figure 6.13: Synchronization based on input data

A possible consequence can be seen in the example in Figure 6.12. During execution of the three COLA clusters depicted at the top of the figure, `Cluster 1` and `Cluster 2` produce inputs for `Cluster 3` and thus have to be executed before `Cluster 3`. In the middle part of the figure, a possible execution on a COLA platform is shown. As can be seen in the figure, the clusters are executed on different

ECUs but the time-triggered schedule makes sure, the inputs for `Cluster 3` have been calculated before its execution. The lower part of the figure shows a possible execution of the clusters on an AUTOSAR platform. In this example, `ECU 1` executes a higher priority task which delays execution of `Cluster 1`. Subsequently, `Cluster 3` lacks one of its inputs when it is executed. Further, when `Cluster 3` is executed the next time, it receives inputs from `Cluster 1` which are possibly outdated, since `Cluster 2` has already been executed another time while `Cluster 1` was delayed.

The above described problem could be solved by another modification to the COLA code generator. As exemplified in Figure 6.13 for `Cluster 3`, a means of synchronization could be realized by checking the respective input data of a cluster. The code generator could insert a checking routine which starts processing of the cluster only if all inputs have been updated. Thus, the dataflow semantics of COLA could be implemented. The exact timing behavior of such a hybrid solution remains, however, unpredictable.

6.4 Chapter Summary

In this chapter we have shown, how the comprehensive system knowledge provided by a COLA model can be employed to produce additional benefit beyond simple deployment of the system. The presented extensions of the deployment concept further improve system quality and provide a possible integration with AUTOSAR.

The concept of model-level debugging enables developers to identify and fix modeling faults at the abstract level of the COLA model. Thus, the implementation of debugging routines at code-level is rendered redundant. Fixes are rather carried out in the model and new code is generated from the altered model. This prevents the occurrence of programming errors during manual coding and guarantees that model and code are always consistent. Further, displaying the results of an execution of the system in the model is often much more convenient compared to e. g. analyzing bus message traces.

Using knowledge about the topology of the target platform and the allocation of tasks, the generation of fault tolerance modes is facilitated. The resulting systems are more reliable, since they feature a redundant allocation of highly safety-critical tasks which are activated in case of a hardware failure.

We concluded the chapter with a concept for the integration of the COLA and AUTOSAR concepts. By implementing the proposed modifications, the execution of clusters modeled in COLA on an AUTOSAR platform is facilitated. The resulting system is a hybrid solution comprising a mixture of tasks modeled in COLA and AUTOSAR. This approach could be used for migrating from an AUTOSAR-based to a COLA-based development.

Chapter 7

Evaluation of Concepts

To prove the practical viability of the COLA approach we completed two case studies during its development. The first case study was implemented early in the development of the COLA concept and focused on the generation of behavioral code from a COLA model. The according demonstrator consisted of a single computing node and any distribution aspects could still be neglected. However, the demontrator already incroporated a middleware for transferring data between the generated application and the hardware devices of the target platform. This middleware provided a fixed API to the code generator for interfacing the hardware platform. The performance of the generated code was benchmarked against a hand-written implementation of the same functionality.

As a step up over the first demonstrator, the second case study was targeted at a distributed solution which takes full advantage of the COLA approach. To this end, the hardware platform consisted of three computing nodes which were connected by a bus system. For the second demonstrator, the application code and platform configuration were generated automatically. The middleware employed on the different ECUs was responsible for transmitting messages between the ECUs and for interfacing the employed hardware devices.

The COLA model and hardware platform of the second demonstrator also served as a test case for the concept of model-level debugging. The logging of data at runtime proved its advantages by helping to identify and remove a few bugs from the application model of the demonstrator.

7.1 Mindstorms Demonstrator

The first demonstrator was implemented using a LEGO® Mindstorms™ controller as hardware platform. It was equipped with the BrickOS[1] operating system. The demonstrator was realized as a small LEGO car which should feature the functionality of an *adaptive cruise control* (ACC) [112]. An ACC is a control device for cars providing the functionality of keeping the car's speed at a value set by the user, while maintaining a minimum distance to the car driving ahead. A picture of the demonstrator can be seen in Figure 7.1.

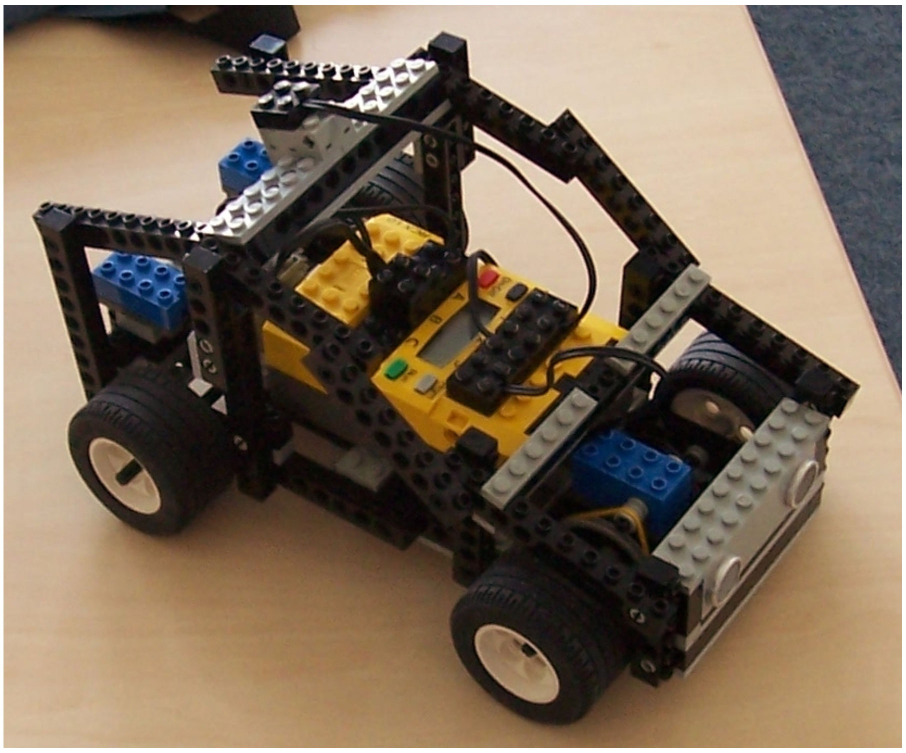


Figure 7.1: The Mindstorms demonstrator

7.1.1 Hardware Platform

The employed LEGO Mindstorms controller — also referred to as *brick* — is based on an Hitachi/Renesas® H8/300 microcontroller which is accompanied by 32 kilobytes of SRAM for storing operating system and programs. In Figure 7.1 the brick is visible as the yellow box inside the model car. Mindstorms controllers provide three input and three output ports for connecting additional devices. Figure 7.2 shows a block diagram of the hardware platform, with the sensors indicated in light

[1]http://brickos.sourceforge.net

gray and the actuators in dark gray, respectively. The mindstorms controller is the white box in the middle of the figure.

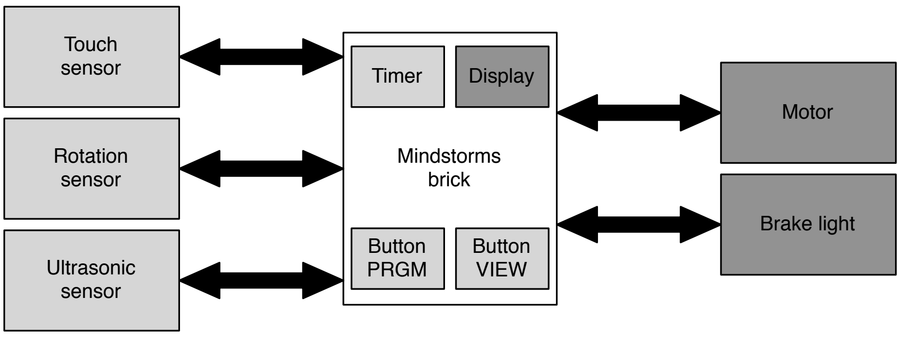


Figure 7.2: The ACC demonstrator hardware platform

For our intended ACC system we used a rotation sensor for measuring rotations of the front axle and an ultrasonic sensor to detect the current distance to the vehicle in front. Further, a touch sensor was used which can be pressed by the user like a conventional push button. In addition the LEGO car was equipped with a motor and breaking lights as actuators. The sensors and actuators were connected to the input and output ports of the Mindstorms controller. Two individually programmed buttons of the brick served as additional input devices. The buttons are labeled `PRGM` and `VIEW`. The display of the brick was used to output information via seven-segment indicators. Finally, the system timer of the brick was used to measure the advance in time. In combination with the count of rotations, the cars speed could be calculated.

Considering the low computational power and small amount of memory, the Mindstorms controller was quite similar to low end automotive ECUs. This property made it a suited platform to test our automatically generated code.

7.1.2 Functionality of the Demonstrator

The intended functionality of the demonstrator includes the possibility to switch the ACC on and off. The display indicates the current ACC state. If the device is turned off, the motor speed set by the user is forwarded to the engine control without any modification. By engaging the ACC, the speed and distance regulation are activated. This includes the measurement and comparison of the pace set by the user and the actual measured car velocity. If the desired user speed s_{user} differs from the actual speed s_{act}, the target speed for the motor control is corrected by $(s_{user} - s_{act})/20$. This results in a speed correction of 5 percent of the difference between actual and desired speed. This regulation is used as long as no object is detected within 35 centimeters ahead of the car. If the distance drops below this threshold, the actual speed is continuously decreased by 5 percent. The minimum distance allowed was set to 15 centimeters. If the actual distance is lower, the car performs an emergency stop. After either reducing speed or coming to a halt, the ACC should speed up the car smoothly again, as soon as the obstacle is out of the critical region. Each time the speed is decreased, brake lights are engaged.

7.1.3 The ACC COLA Model

The implementation of the described functionality is guided by the hardware available. The used controller offers only two buttons available for control actions to the operator. Additionally, three sensor and three actuator ports are present. For the demonstrator we use the two controller buttons for setting the desired user speed. A touch sensor is employed to switch the ACC on and off. The remaining two sensor ports are used to connect a rotation and an ultrasonic sensor for measuring the current speed and distance, accordingly. The motor is connected to one of the actuator ports, while a second port was utilized to connect the brake lights.

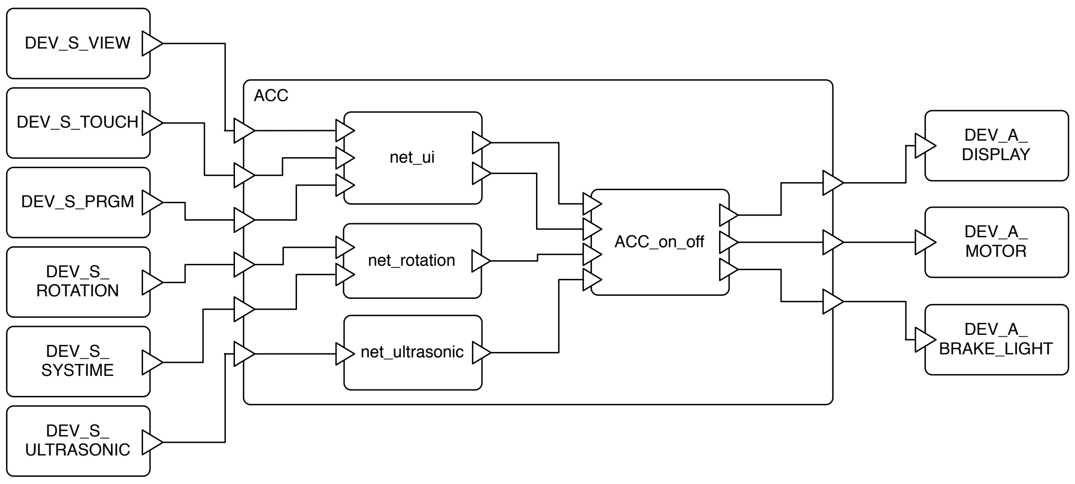


Figure 7.3: The ACC COLA model

The top-level network of the ACC COLA model is shown in Figure 7.3. Each of this network's ports is connected to a source or sink which are mapped to one of the sensors and actuators used for the ACC system. The source and sink blocks are indicated by a small, black triangle in Figure 7.3. Further, the blocks' names carry a prefix of either `DEV_S_` or `DEV_A_` to make clear they are sensor or actuator devices, respectively. The connected sensors are in particular: the brick's buttons `DEV_S_VIEW`, `DEV_S_PRGM`, the rotation sensor `DEV_S_ROTATION`, the system clock `DEV_S_SYSTIME`, the touch sensor `DEV_S_TOUCH`, and the distance sensor `DEV_S_ULTRASONIC`. The actuators connected to the brick are the controller's display `DEV_A_DISPLAY`, the motor `DEV_A_MOTOR`, and the brake lights `DEV_A_BRAKE_LIGHT`. As described in Section 5.4 these sources and sinks are interfaced using the middleware calls `mw_receive()` and `mw_send()`. For example, a channel connected to `DEV_A_MOTOR` would be transformed into `mw_send(&s_mot_5, sizeof(&s_mot_5), 13)` where `s_mot_5` is the calculated output for the motor speed. The address `13` is the numeric middleware address assigned for this actuator. The middleware forwards the receive and send calls to hardware driver calls provided by BrickOS, which in turn provides sensor values or modifies some actuator state.

Since this case study was intended to be executed on a single node, i.e., a mindstorms controller, we chose to put all parts of the model which should be implemented in software into a single cluster. To this end, the network `ACC` was defined to represent the only working cluster. As no distribution is possible using a single node, the generation of a single task results in the fastest system possible. This avoids the — in this case — unnecessary middleware communication overhead of a distributed solution. Consequently, the comparison of runtimes against a hand-written version of the ACC presented in Section 7.1.4 was facilitated, as the manual implementation was also realized as a single task. Thus, the resulting runtimes are less influenced by OS and communication overhead, and the results rather depend on efficiency of the compared application code.

For a distributed system, a more fine-grained clustering might be favorable.

7.1.4 Benchmarking the ACC Code

When working with generated code, efficiency aspects surely play an important role. Using any valid tricks the programmer is aware of, hand-written code may be considerably smaller and faster. Consequently, the benefit of automatic code generation is rather its deterministic result. If behavioral correctness of the model has been ensured using techniques such as model checking, a code generator may produce equivalent correct code. Coding errors like unintentional casts, wrong pointer arithmetic and the like are avoided. Finally, a major fraction of the potential performance drawbacks are negligible due to optimizations performed by the compiler.

To get an idea of the performance of the generated ACC code, we compiled it successively using the optimization levels O1, O2 and O3 of the GNU C compiler and checked the runtime results against the ones of a hand-coded version of the ACC. To give an impression of the quality of our manually coded version, we give the lines of code (LOC) as metrics. The hand-coded version fulfills the same functionality using 75 LOC, while the code generator produced 249 LOC for the modeled ACC. The resulting binaries were compared regarding their execution times. To minimize errors in the measurement of the running period induced by interrupts, context switches, etc., the ACC algorithm was called consecutively 100 times. We ran this benchmark 20 times for each optimization level.

The averaged resulting times are given in Figure 7.4. The third bar in the diagram, named *generated (static fct's)*, indicates the values for a modified version

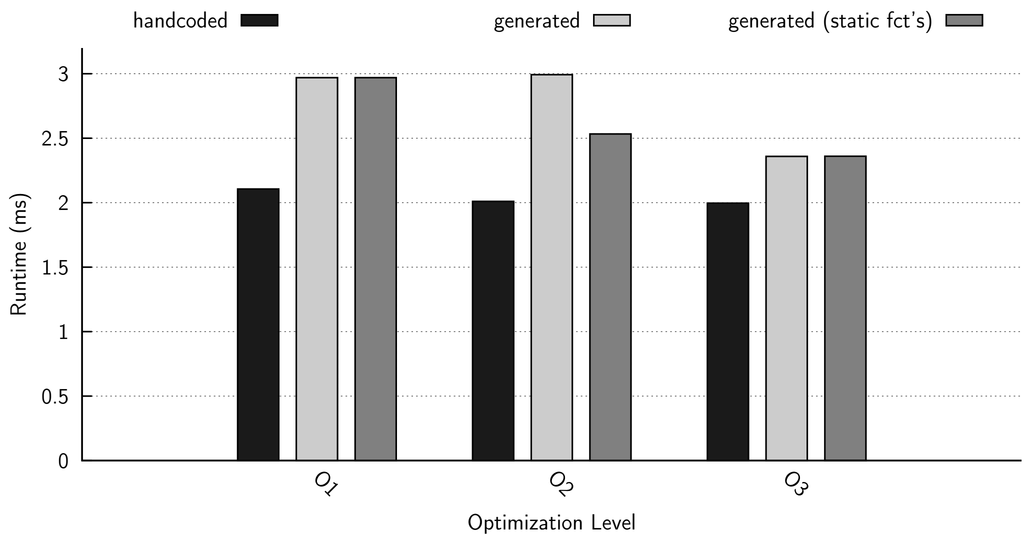


Figure 7.4: Benchmarking runtime results

of the generated code, where all functions are declared static. As this allows the compiler to disregard the use of the functions from outside the binary, the function calls can be replaced by the functions' respective implementation. As a result, there is no jump to fulfill and, thus, the execution time is decreased.

As can be seen in Figure 7.4, all versions of the code benefit from an increase in the optimization level of the compiler. Obviously, the use of static functions has most effect in case of the O2 optimization level. The measurements show that the execution time of our generated code isn't too far from the version implemented by hand. Especially when using the higher optimization levels of the compiler. There the generated code benefits even more than the hand-coded alternative. At maximum optimization level, i.e., O3, the generated code actually reaches almost the execution times of the handwritten version.

7.2 Multi-Node Demonstrator

The second demonstrator was intended to show the usability of the COLA approach for a distributed system. To this end, it was based on a hardware platform that incorporated three processing nodes. The sensors and actuators used for this platform where connected to the different processing nodes. The nodes themselves where interconnected by a bus system. All these hardware devices where built into a 1:10 scale model car, which can be seen in Figure 7.5.

The idea for the case study was to build a model car featuring a function also available in real cars. We decided to implement an autonomous parking system based on several distance sensors. In addition, the system should be controllable

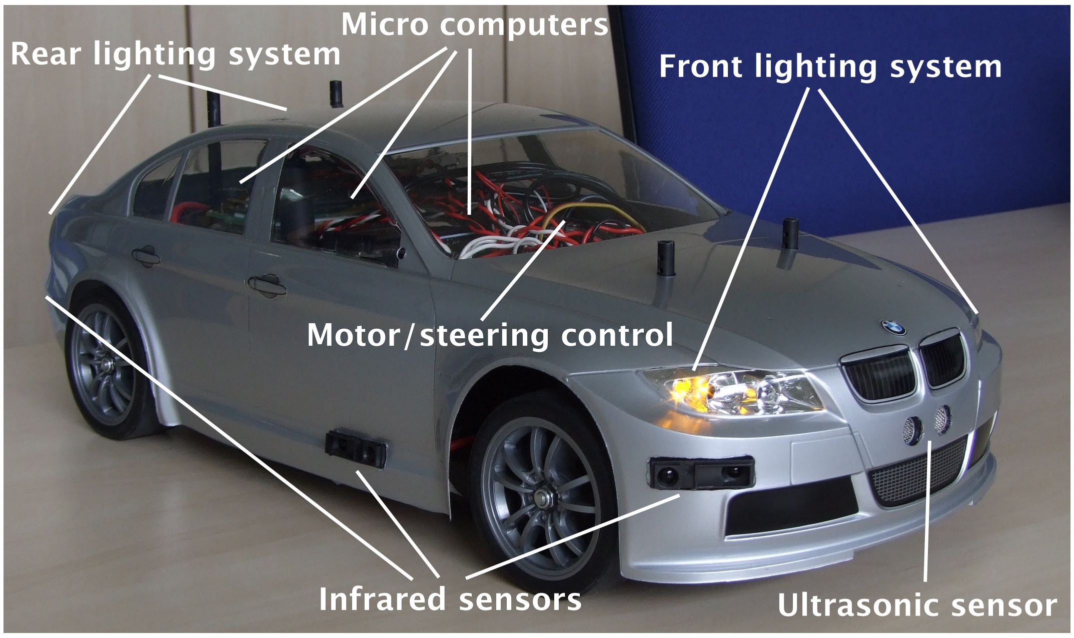


Figure 7.5: Multi-node demonstrator

manually. A cell phone which was connected to the car via Bluetooth was used as the remote control.

For the second demonstrator, prototypical implementations of the deployment tools described in Chapter 5 where available. Further, a graphical COLA editor was available for designing the Functional and Technical Architectures. This eased work a lot compared to the first demonstrator, where the model had to be specified in textual representation. The implementation of the multi-node demonstrator could be generated automatically using the tool prototypes.

7.2.1 Hardware Platform

The model car shown in Figure 7.5 was equipped with three Gumstix® micro computers connected by an Ethernet network. The micro computers, which represent the electronic control units (ECU) of the demonstrator, are based on a Marvell® PXA255 processor. They feature 64 megabytes of RAM and 16 megabytes of flash memory. For communication, each of the Gumstix was outfitted with an add-on board containing the Ethernet controller. Further, the employed `Gumstix connex 400xm-bt` are equipped with a Bluetooth controller. Figure 7.6 shows a block diagram of the hardware platform.

As can be seen in Figure 7.6, a number of input devices, indicated in light gray, and output devices, shown in dark gray, was connected to the ECUs. Three infrared and one supersonic sensor were used to measure distances. The supersonic sensor

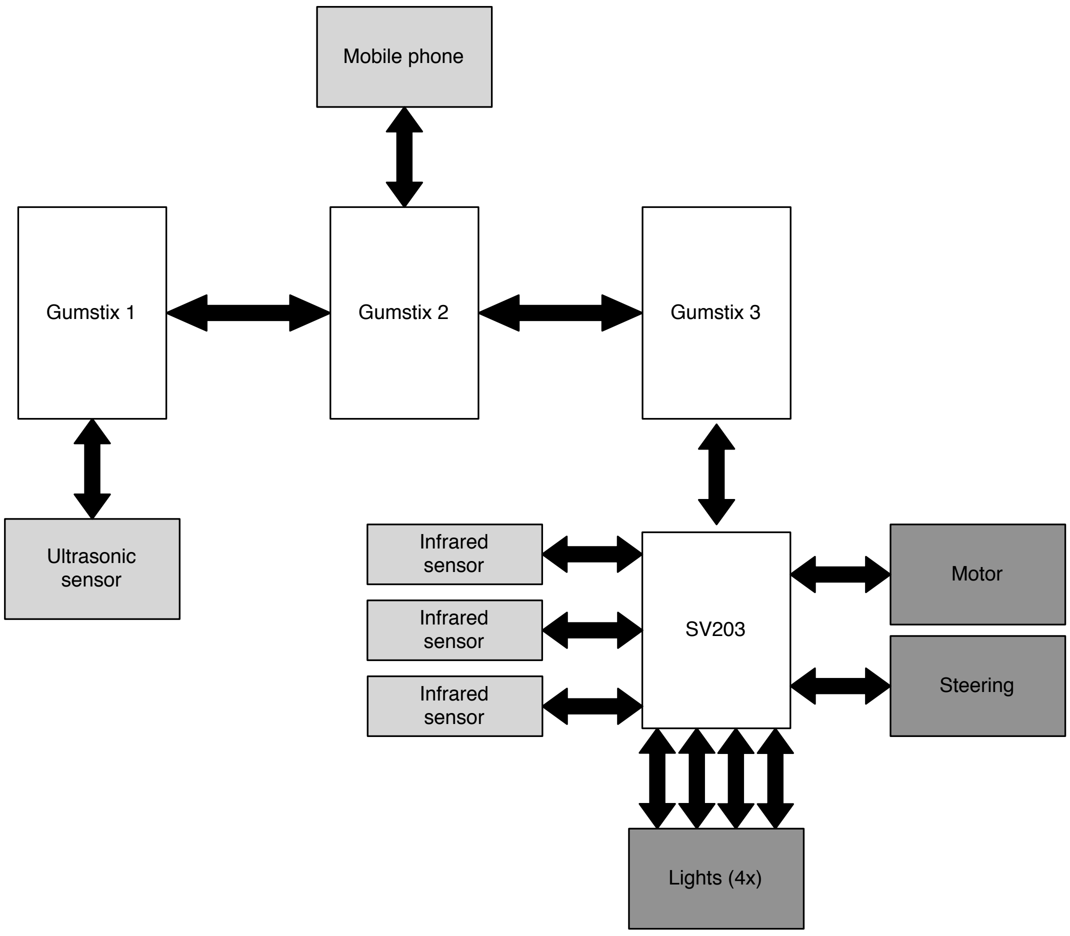


Figure 7.6: The multi-node demonstrator hardware platform

was directly connected to one of the Gumstix using a I^2C connection. For connecting the infrared sensors, an additional interface card — denoted as `SV203` — which incorporated an analog digital converter was employed. The interface card itself was connected to another Gumstix by means of a serial connection. The Bluetooth controller of the third Gumstix was used to connected to the cell phone remote. The model car's motor and steering, as well as indicator, reversing, and breaking lights represent the actuators of the system. They were also accessed using the SV203 interface card. Due to the fact that the mentioned sensors and actuators were connected to different ECUs, the system had to be executed in a distributed manner.

Regarding the software platform, an implementation of the COLA middleware was employed for data exchange and clock synchronization. Xenomai [51] served as the real-time operating system for the ECUs.

7.2.2 Functionality of the Demonstrator

The multi-node demonstrator features three different operating modes. It can either be in *manual control*, *parking*, or *side distance control* mode. The remote control may be used to switch between these modes. We will give a short explanation of the modes in the following:

Manual control mode: In manual mode the model car can be controlled using the mobile phone as remote. The user may modify the speed and direction of the car using the directional pad of the phone. To avoid collisions, the car uses the ultrasonic distance sensor at the front. The car initiates an emergency stop when reaching a given minimum distance to obstacles. Indicator lights flash as soon as the car is instructed to turn.

Parking mode: The goal of the system in parking mode is to search and find a suitable parking space and use it for parallel parking. For our model car we chose to drive along a wall substituting a row of parked cars. The model car is intended to drive in parallel to this wall and detect gaps — like an open door — which are big enough to park. As soon as such a gap is found, the car shifts into reverse and carries out a reverse parking maneuver.

For the parking mode three infrared distance sensors are used in addition to the supersonic sensor. As can be seen in Figure 7.5, the infrared sensors are placed at the right side, the right front corner, and the back of the car. As soon as the parking mode is activated, the car switches to a low, constant speed and uses the four sensors to search for the wall and a suitable parking space. Controlling the motor and steering settings the car then parks into the space. The lighting system is engaged accordingly, showing turn and reverse indicators as well as braking lights.

Side distance control mode: The side distance control mode uses part of the functionality of the parking mode. Just as in parking mode, the distance sensors are used to drive in parallel to the wall. But compared to the parking mode, no parking spaces are searched. Instead, the car tries to keep the distance to the wall consistent at all times. Further, in side distance control mode, the speed may be modified by the user. Using the ultrasonic sensor at front, obstacles are detected and collisions avoided.

To switch between these modes, the number pad of the phone is used. As a safety switch, the 0-key may be pressed at any time. This instructs the car to execute an emergency stop, regardless of the current mode. We will give some figures about the COLA model for the described functionality in the following.

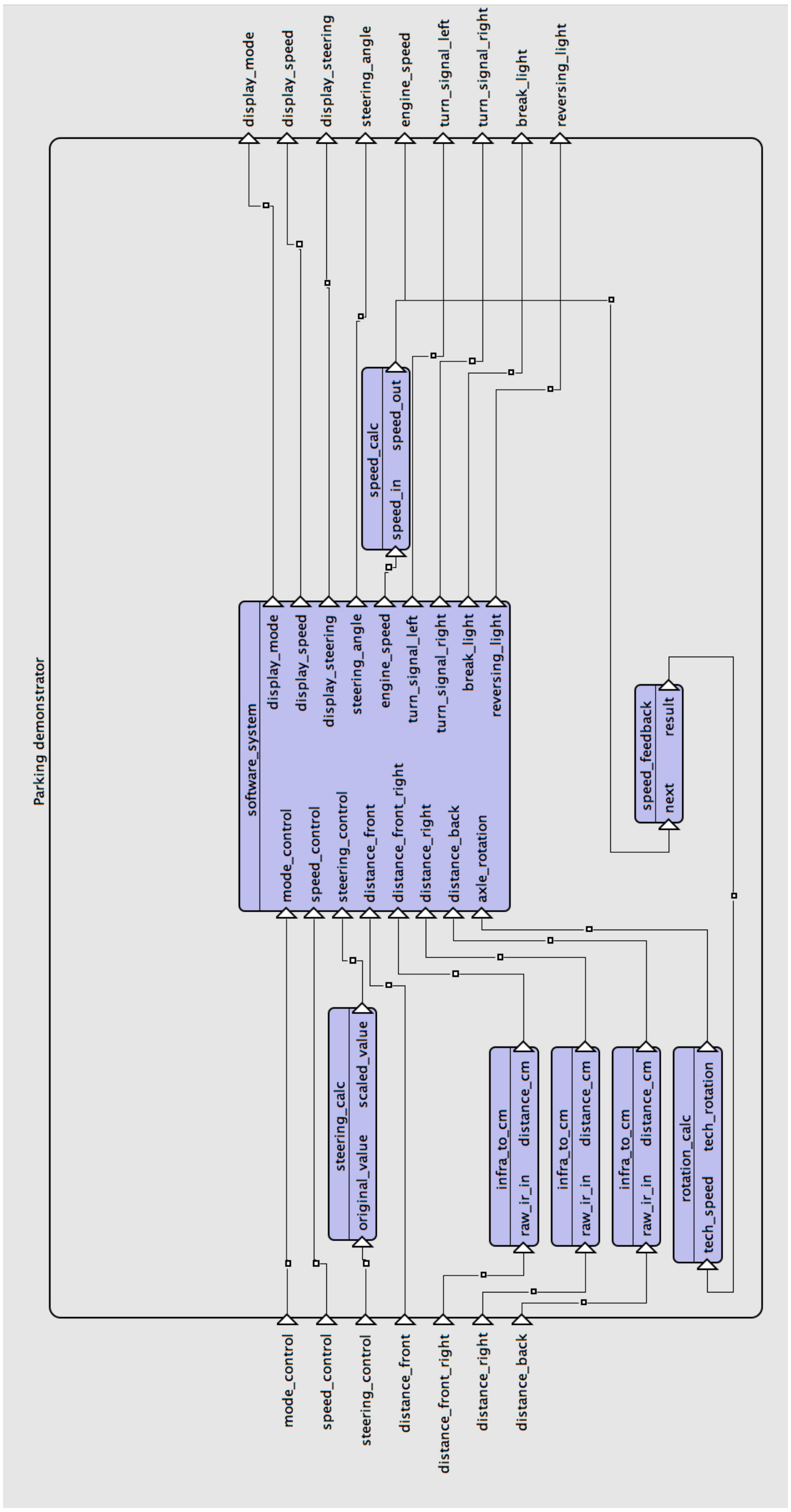


Figure 7.7: The parking demonstrator COLA model

7.2.3 The Parking Assistant COLA Model

Figure 7.7 shows a network of the multi-node demonstrator COLA model. In this network most of the clusters for the multi-node demonstrator are defined. The overall model incroporates 37 automata, consisting of 100 states, and 225 networks. For the deployment, these COLA units were partitioned into one mode cluster and ten working clusters allocated to the different Gumstix computers.

The mode cluster is named `software_system` and is visible in the middle of Figure 7.7. It is implemented by an automaton consisting of three states, which correspond the three afore mentioned operating modes. Depending on the state of the mode cluster, which is selected according to the input from the remote, a working cluster implementing the respective mode is chosen for execution. The remaining seven working clusters can be seen next to the mode cluster in Figure 7.7. These working clusters are responsible for data pre- and post-processing of sensor and actuator data, as well as a feedback loop.

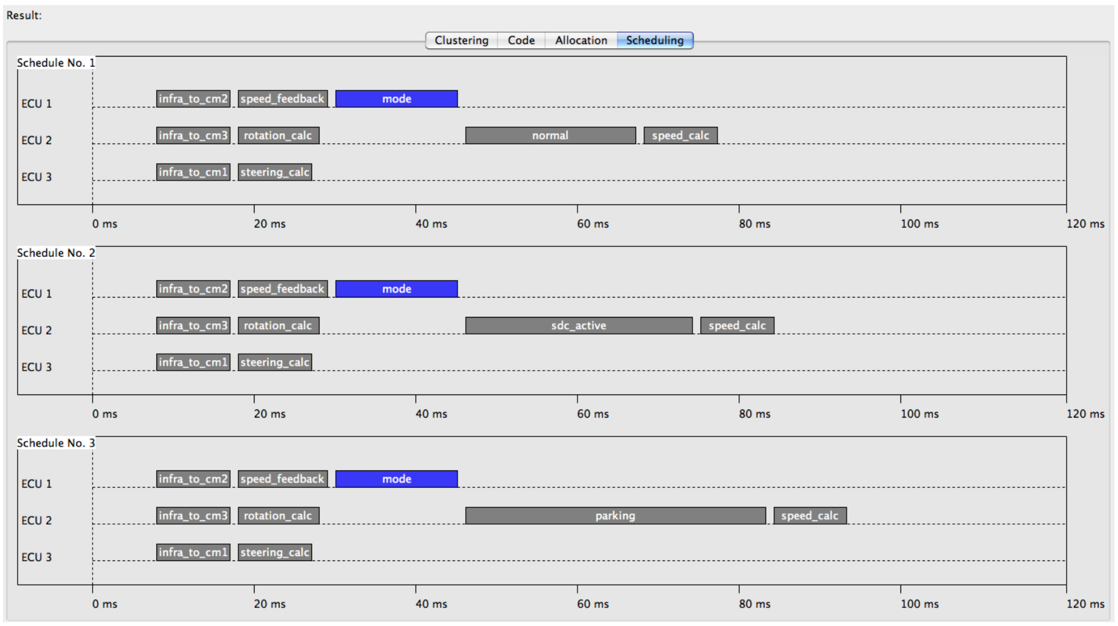


Figure 7.8: Scheduling result for the parking demonstrator

Using the deployment tools, the COLA model was translated into executable code and configuration data for the multi-node demonstrator. After producing C code, the allocation of all clusters was calculated. A feasible schedule was then generated. In the deployment tool a graphical representation of the scheduling result is available. The result for the multi-node demonstrator can be seen in Figure 7.8. Since the mode automaton features three states, three different schedules are created. These schedules are all identical until the mode automaton, shown in blue, is executed. Depending on its result, either the manual control (`normal`), parking (`parking`), or side distance control mode (`sdc_active`) is executed. The system switches these schedules at runtime, as described in Section 5.6. The other tasks shown in the schedule correspond to the equally named clusters from the COLA model in Figure 7.7.

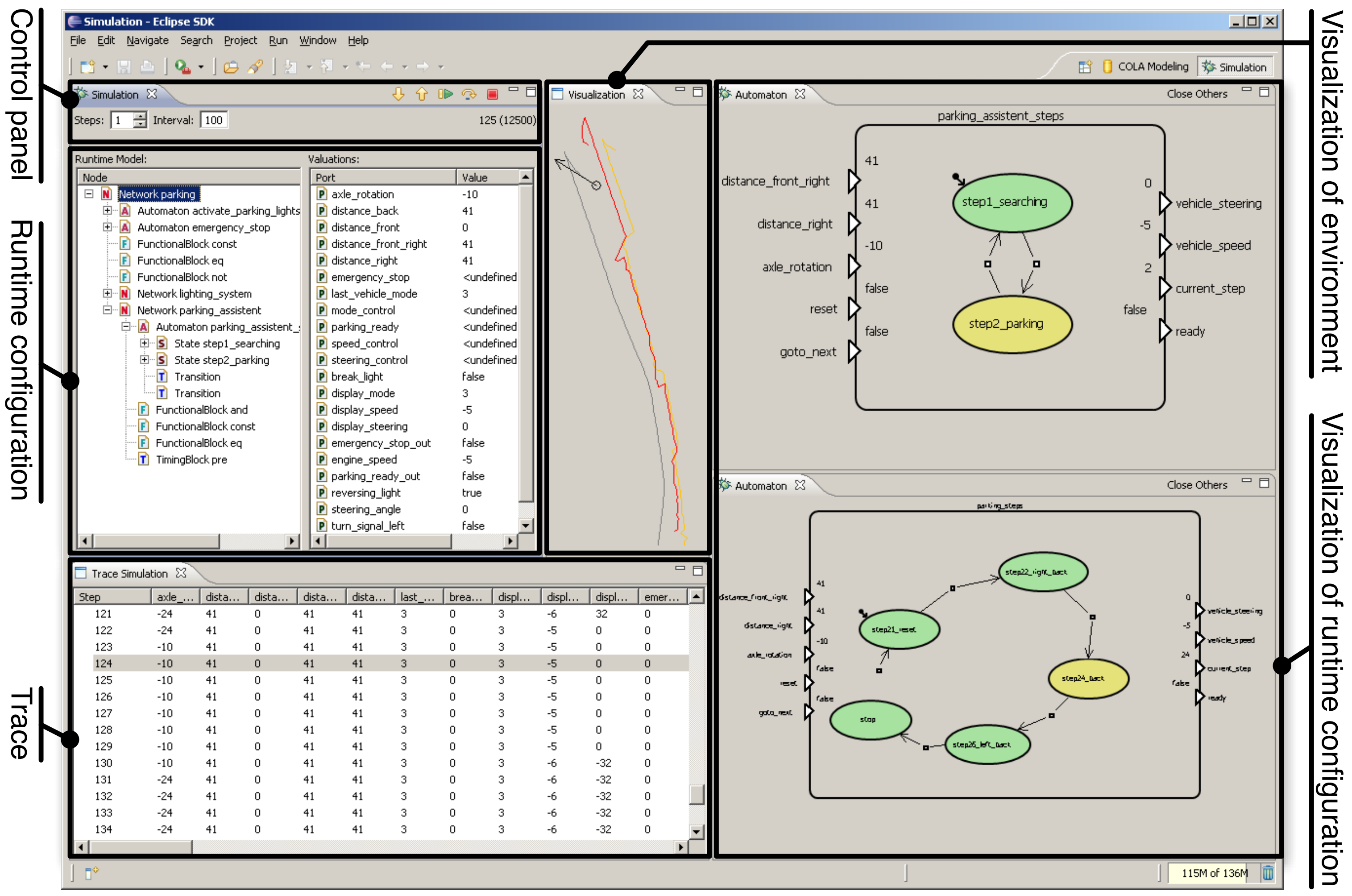
After some debugging of the functional model, using the model-level debugging approach, the demonstrator showed the desired behavior. We will describe our experiences with the logging of runtime data in the demonstrator, next.

7.3 Runtime Data-Logging

During development of the afore described multi-node demonstrator, the tracing of actual data proved to be very helpful. An early version of the modeled system did not behave as expected. Using model-level debugging, we were able to identify several small faults in the model. One case, for example, was related to some flipped channels in the model, thus forwarding data of the employed sensors to the wrong inputs. Additionally, the generated values of the used infrared and ultrasonic sensors were not as constant as assumed in the model. There where steps in the measured distance values, even though the actual distance did not change. By tracking actual data in the simulator, we were able to identify the problem and add filters for these input values to the system model. The corresponding `parking` cluster contained 115 units and was traced for 147 invocations, corresponding to 14.7 seconds. The generated trace file was 45 KByte in size. So the tracing of real-time data showed to be a valuable addition, while producing moderate memory consumption.

Figure 7.9 shows a screenshot of the COLA simulator during execution of a trace from the multi-node demonstrator. The `control panel` on the left gives the user the possibility to start, pause, and stop the replay of the loaded trace. In the `runtime configuration` window a textual view of the model elements and their current input and output values can be seen. An overview of all values captured on the target platform can be seen in the `trace` window. On the right side of the simulator interface a `visualization` of the runtime configuration is given. Here, the input and output values are annotated next the according ports. The currently active state of an automaton is marked yellow.

To give the user an impression of the target system's `environment` during execution, another visualization window has been added to the simulator. In Figure 7.9 it shows the values of the demonstrator's side distance sensors as red and yellow lines, and its trajectory as a black line. The current position of the car is shown as a small circle with an arrow indicating the current heading. This visualization of environment is, of course, specific for every platform and has to be implemented, accordingly. However, the implementation of the simulator makes this task easy because of its clear structure, as has been described in [71].



Steps: 1 Interval: 100 125 (12500)

Runtime Model:

- Network parking
  - Automaton activate_parking_lights
  - Automaton emergency_stop
  - FunctionalBlock const
  - FunctionalBlock eq
  - FunctionalBlock not
  - Network lighting_system
  - Network parking_assistent
    - Automaton parking_assistent_
      - State step1_searching
      - State step2_parking
      - Transition
      - Transition
    - FunctionalBlock and
    - FunctionalBlock const
    - FunctionalBlock eq
    - TimingBlock pre

Valuations:

Port	Value
axle_rotation	-10
distance_back	41
distance_front	0
distance_front_right	41
distance_right	41
emergency_stop	<undefined
last_vehicle_mode	3
mode_control	<undefined
parking_ready	<undefined
speed_control	<undefined
steering_control	<undefined
break_light	false
display_mode	3
display_speed	-5
display_steering	0
emergency_stop_out	false
engine_speed	-5
parking_ready_out	false
reversing_light	true
steering_angle	0
turn_signal_left	false

Trace Simulation

Step	axle_...	dista...	dista...	dista...	dista...	last_...	brea...	displ...	displ...	displ...	emer...
121	-24	41	0	41	41	3	0	3	-6	32	0
122	-24	41	0	41	41	3	0	3	-5	0	0
123	-10	41	0	41	41	3	0	3	-5	0	0
124	-10	41	0	41	41	3	0	3	-5	0	0
125	-10	41	0	41	41	3	0	3	-5	0	0
126	-10	41	0	41	41	3	0	3	-5	0	0
127	-10	41	0	41	41	3	0	3	-5	0	0
128	-10	41	0	41	41	3	0	3	-5	0	0
129	-10	41	0	41	41	3	0	3	-5	0	0
130	-10	41	0	41	41	3	0	3	-6	-32	0
131	-24	41	0	41	41	3	0	3	-6	-32	0
132	-24	41	0	41	41	3	0	3	-6	-32	0
133	-24	41	0	41	41	3	0	3	-6	-32	0
134	-24	41	0	41	41	3	0	3	-6	-32	0

Figure 7.9: The COLA simulator executing a trace

7.4 Chapter Summary

In this chapter we have presented two case studies which have been implemented to prove the usability of the COLA approach.

The single-node demonstrator contained rather little functionality and its main purpose was to showcase the generation of application code from a COLA model. For the multi-node demonstrator with its bigger processing capacity, the more complex task of autonomous driving and parking could be realized. This case study posed the challenge to design a larger scale COLA model, which was valuable to verify the scalability of COLA for larger systems. Further, the execution platform of the multi-node demonstrator was composed of three processing units. Thus, the distribution aspects of the COLA approach, like allocation, scheduling, and platform configuration could be tested. After some refinement of the prototypical COLA tool-chain and the underlying processes, the case study could be automatically supplied with software as desired.

Finally, the concept of model-level debugging could be tested using the multi-node case study. The concept demonstrated its value for the COLA process during bug fixing of the case study's application model.

Chapter 8

Conclusions

In this chapter we present a summary of the concepts and ideas presented in this thesis. We will recapitulate in this context the main contributions of our work for the COLA approach. Further, we want to discuss how the COLA approach could be integrated into today's development process. Finally, we want to present some perspectives, how automotive development in general — and the COLA approach in particular — might be improved for use with future automotive systems.

8.1 Summary

In this thesis we have shown the necessary steps to automatically generate an executable automotive system from a COLA model. The COLA approach is a novel MDD concept for the development of reliable automotive system software. In contrast to other MDD concepts it features integrated modeling. Hence, it can be employed throughout the entire development process of a distributed automotive system [55]. Using the COLA language, transitions between process steps are simplified due to the employment of a single modeling language. This absence of gaps between different tools — which are the norm nowadays — avoids errors which arise from porting data between tools. As a result, the systems designed with COLA are of higher quality. This quality can be improved even further, using model checking for COLA models. Due to its accurately defined semantics, automated checks of systems modeled in COLA are made possible.

In order to retain the model's quality for the actual system, the concepts presented in this thesis have proved their utility during implementation of the case studies. Using the described deployment process, an executable distributed system is generated, which can be assumed to be correct by design. The automatic deployment concept includes *generation of application code*, *allocation* and *scheduling* of tasks, and *configuration* of the target platform. This high degree of automation is backed by a custom *platform concept*. Further, all *platform information* relevant for the success of the automatic deployment may be modeled in COLA.

Using the wealth of information available in a COLA model additional benefits can be derived. By combining all information available about task distribution, execution timing, and communication, runtime data of the actual system may be mapped back to the model. These data may be replayed in the model simulator which facilitates *debugging* of the system at model-level. Another benefit is the possibility to automatically generate *fault-tolerant systems* from the model.

We will recapitulate the benefits of our main contributions in the following:

Platform concept: In order to be suited as target for automatic deployment, the platform has to fulfill some basic conditions. For the deployment of a COLA model we proposed the use of a time-triggered platform. This includes the employment of a polling mechanism for transmitting data between sensors and actuators, and the application processor of each computing node. Further, the operating system has to provide non-preemptive scheduling of tasks at predefined points in time. Thus, system-wide scheduling plans can be executed synchronously on all nodes of the distributed hardware platform.

On top of the operating system, a custom middleware has been designed for the COLA deployment [53]. This middleware provides synchronization of a global clock being used by the respective schedulers to determine the reference time. In addition, the middleware features transparent communication for the generated application code. Using the middleware's API, a task is able to exchange data with other tasks as well as with sensors and actuators. The known API of the middleware simplifies generation of the application code.

We have proposed a graph structure, called cluster dependency graph, which is used to capture the data dependencies between tasks of the system [84]. To this end the data paths modeled in COLA are analyzed. As a result, the graph contains information about the execution order of tasks implied by the model. Moreover, the graph is used to assign middleware addresses for use by the tasks.

Platform modeling: A COLA model of the hardware platform is used as a guidance for deployment. The outlined concept for modeling the platform is suited to capture all information needed for this task. To this end the COLA Hardware Architecture gives an abstract description of the platform's topology consisting of computing nodes, buses, sensors, and actuators. By comprising information about properties and capabilities of these components, suitable allocation and scheduling solutions can be derived.

Generation of application code: Our application code generator produces C code files for all tasks designed in the COLA model [58, 59]. Calls to the middleware are used for communication with other tasks or the sensors and actuators of the underlying platform. The code generator inserts these calls using the addresses stored in the cluster dependency graph. In addition, the code generator inserts commands to load and store the internal state of tasks via the middleware.

The result of this code generation is a set of code files comprising the application tasks of the system. The code is independent of the location of its execution, thanks to the use of our middleware. Hence, the assignment of tasks to computing nodes can be defined in a subsequent step.

Allocation and Scheduling: Our deployment concept features automatic allocation and scheduling of generated tasks to computing nodes of the target platform. This is achieved by calculating worst-case execution times and memory consumption for every generated task [129]. In addition, non-functional requirements may be specified for the task. Using these information and matching them with the platform model, a suitable allocation may be calculated [85]. When an allocation has been found, it is checked for schedulability. This process is repeated until a valid combination of allocation and system schedule has been found. The developer is informed if the allocation or scheduling problem can not be solved for the given combination of software and hardware model.

During the allocation step the cluster dependency graph is updated with the calculated mapping of tasks to computing nodes. The scheduling algorithm uses this information to derive end-to-end runtimes for applications consisting of several communicating tasks.

Automatic platform configuration: Existing code generation concepts are usually limited to producing a centralized system. In contrast, the concept presented in this thesis, aims at the generation of distributed systems. To this end, generation of application code is not sufficient. The configuration of the platform also has to be addressed to provide integration of the tasks allocated to different nodes of the system. In this work we have shown how to derive a platform configuration from a COLA model.

Using the proposed platform configuration concept, it is not only possible to execute a fixed set of tasks. Rather, operating modes may be defined in the model, which are changed synchronously in the actual system [56].

These changes are achieved by using the middleware to communicate changes of the current operating mode. The schedules of all computing nodes are then changed synchronously, thus activating another set of tasks for the new operating mode.

Model-level debugging: As an addition to the basic deployment concept, we have proposed model-level debugging [54]. This add-on facilitates the capturing of input and output data as well as the internal states of tasks on the target platform. Hence, actual runtime data are available for debugging. Our concept allows the import of these data into the COLA model simulator [71]. Consequently, an execution of the system on the target platform can be replayed in the simulator.

The concept is an addition to classical remote debugging and low-level debugging mechanisms. If used consistently, error-prone manual changes to the code can be avoided. Further, if debugging is carried out in the model, target system and model are always kept synchronous. This eases the implementation of extensions and the re-use of the model.

Fault tolerance modes: We have outlined another extension to the deployment concept in form of generating fault tolerance modes [57]. According to the concept, a fault tolerance mode is essentially a special case of an operating mode.

To switch into a fault tolerance mode if necessary, the middleware continuously monitors the communication between the computing nodes of the system. If a node fails to deliver an anticipated message, the middleware assumes this node to have failed. Hence, it initiates a change into the according fault tolerance mode. In that mode, the safety-critical tasks which have previously been executed on the now failed node are relocated to another node. This can be achieved by allocating the tasks redundantly during deployment and activating a suitable schedule at runtime. The developer may specify which tasks are safety-critical in the model. As a result, during deployment these tasks are allocated and scheduled redundantly.

Using the proposed deployment concepts leads to a higher quality system which complies with the COLA specification. While the integrated character of COLA helps avoiding errors during system specification, the deployment concept is suited to prevent errors during implementation of the modeled design.

8.2 Discussion

Talking of MDD for automotive systems, one concept that comes to mind naturally is AUTOSAR. Just like the COLA approach, AUTOSAR provides a means of modeling an automotive system and proposes a platform concept as well. AUTOSAR differs, however, from the COLA approach regarding its main purpose. AUTOSAR modeling is limited to the design of a software system's architecture, while COLA also facilitates modeling of requirements and functionality. This limitation of AUTOSAR has one reason: it is targeted at today's typical development process. This process is based on a distribution of labor between automotive OEM and suppliers.

While the OEM specifies the desired automotive system, suppliers develop black boxes of software and hardware with the specified functionality. These black boxes are then integrated by the OEM into a complete system. The intellectual property contained in the ECUs remains in the majority of cases with the supplier. This lack of knowledge makes integration for the OEM tedious and time consuming. To ease the integration of subsystems for the OEM, AUTOSAR enables the modeling of interfaces between the different components. This results in better compatibility of the different subsystems and, hence, easier integration. Further, as several OEMs adopted the AUTOSAR standard, it allows the suppliers to deliver an already implemented function to several OEMs without major changes.

The COLA approach, in contrast, does not yet adhere to the described development process. To take full advantage of the COLA approach, it is necessary to have a complete model of the overall automotive system including its functionality. As a benefit model-checking and analyses necessary for automatic deployment can take place. If the COLA concept would be used in the relationship described above, all parties would have to integrate their work into a single COLA model. But as a side effect, the suppliers' knowledge would be disclosed to everyone who has access to the model. This distribution of information is, of course, neither desired by the OEM nor the suppliers. Thus, the use of COLA would either require a change in the relationship between OEM and suppliers, or the COLA approach needs to be modified to fit the actual development process.

As a solution to the described issue, we propose the use of a central database for the COLA model. This database should restrict access of all parties to only their part of the system. Confidentiality of the data could be ensured, if the database would be hosted by a dedicated third party. This carrier is only responsible for storing data and not involved in the system design. The COLA tools for model checking, simulation, and deployment are then also executed on the central database. Thus, they have access to the entire system model. Results produced by the tools are then forwarded by the carrier to the respective developer.

Another open issue in the COLA approach is the quality of the platform software. With all application and configuration code generated, operating system, drivers, and middleware remain the only parts of the system which are coded manually. Compared to the applications, these parts remain relatively stable over several models or generations of cars. Still, special care has to be taken during their implementation and thorough testing is advisable.

8.3 Perspectives

The automotive industry has experienced, and still is experiencing, a huge shift towards the use of distributed embedded computing systems. Today, these systems are responsible for a good amount of innovations in automobiles. This situation demands automotive OEMs to become — at least to a certain point — software producers. Otherwise exclusivity of new functionality is difficult to guarantee. The COLA approach could be a welcome solution to this requirement. Using its concepts, automotive systems can be designed which are at the same time capable and of high quality. It is well suited to enrich the requirements modeling already carried out by the OEM with a functional design of the system.

To simplify development using COLA, the definition of ready-to-use networks and automata would be desirable. These networks and automata could implement functions which are used over and over again in control engineering. Typical examples are data conversions, PID controllers, etc. Providing such units in form of a library would speed up the development process.

Reducing the number of computing nodes in automotive systems seems to be only a question of time. The resulting hardware platform will consist of few high-performance nodes and high-speed busses connecting them. Guaranteeing deadlines in face of a huge number of tasks and messages competing for resources demands tool support for allocation and scheduling. Comprehensive modeling concepts like COLA are able to meet this demand.

The availability of an overall system model helps designing comprehensive functionality. Saving energy is a recent topic for automotive systems. This can only be achieved, if all nodes of the system cooperate in switching to low energy states or shutting down parts of the network. Comprehensive knowledge about the system is a must to avoid faults in such an energy-aware system. Using the COLA deployment, low energy operating modes could be defined, which are then generated from the model automatically. The current deployment approach is already able to adapt the executed schedule accordingly. At platform level, the operating system and middleware would be required to provide a mechanism for shutting down parts of the hardware. Then the change into an energy saving mode could also trigger the necessary power saving mechanisms of the underlying hardware.

Another possible extension of the deployment concept would be a wider coverage of code generation. As mentioned above, the manually coded parts of the system are prone to contain errors. From the information available in a COLA model, it would be reasonable to generate the middleware. As a result, some flexibility is lost regarding the addition of applications in an already deployed system. On the other hand the resulting code is eventually smaller because it only contains instructions for the respective node the middleware instance is generated for. And, of course, the risk of coding errors is decreased.

Abbreviations

ABS	Anti-Lock Braking System
ACC	Adaptive Cruise Control
AST	Abstract Syntax Tree
AUTOSAR	Automotive Open Systems Architecture
CAN	Controller Area Network
CDG	Cluster Dependency Graph
CNI	Communication Network Interface
EAST-ADL	Electronics Architecture and Software Technology - Architecture Description Language
ECU	Electronic Control Unit
FA	Feature Architecture
ILP	Integer Linear Programming
IMA	Integrated Modular Avionics
LA	Logical Architecture
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MDD	Model-Driven Development
MISRA	Motor Industry Software Reliability Standard
MOST	Media Oriented Systems Transport
NFR	Non-functional Requriement

OEM	Original Equipment Manufacturer
OIL	OSEK Implementation Language
PIM	Per Instance Memory
RTE	Runtime Environment
SALT	Smart Assertion Language for Temporal Logic
SWC	AUTOSAR Software Component
TA	Technical Architecture
TDMA	Time Division Multiple Access
TTA	Time-Triggered Architecture
TTC	Time-triggered Co-operative
TTP	Time-Triggered Protocol
UML	Unified Modeling Language
VFB	Virtual Functional Bus
WCET	Worst-case Execution Time

List of Figures

List of Tables

Listings

Publications

Kugele2007	Stefan Kugele, Michael Tautschnig, Andreas Bauer, Christian Schallhart, Stefano Merenda, Wolfgang Haberl, Christian Kühnel, Florian Müller, Zhonglei Wang, Doris Wild, Sabine Rittmann, and Martin Wechs: COLA – The Component Language. In: *Technical report*, TUM-I0714, Institut für Informatik, Technische Universität München.
Haberl2008a	Wolfgang Haberl, Michael Tautschnig, and Uwe Baumgarten: Running COLA on Embedded Systems. In: *ICSE'08: Proceedings of the IAENG International Conference on Software Engineering*, Hong Kong, China, March 2008.
Wang2008a	Zhonglei Wang, Wolfgang Haberl, Stefan Kugele, and Michael Tautschnig: Automatic Generation of SystemC Models from Component-based Designs for Early Design Validation and Performance Analysis. In: *WOSP'08: International Workshop on Software and Performance*, Princeton, USA, June 2008.
Kugele2008a	Stefan Kugele and Wolfgang Haberl: Mapping Dataflow Dependencies onto Distributed Embedded Systems. In: *SERP'08: International Conference on Software Engineering Research and Practice*, Las Vegas, USA, July 2008.

Haberl2008b	Wolfgang Haberl, Uwe Baumgarten, and Jan Birke: A Middleware for Model-Based Embedded Systems. In: *ESA'08: International Conference on Embedded Systems and Applications*, Las Vegas, USA, July 2008.
Haberl2008c	Wolfgang Haberl, Michael Tautschnig, and Uwe Baumgarten: From COLA Models to Distributed Embedded Systems Code. In: *IAENG International Journal of Computer Science*, Volume 35 Issue 3, pges 427-437, International Association of Engineers, September 2008.
Kugele2008b	Stefan Kugele, Wolfgang Haberl, Michael Tautschnig, and Martin Wechs: Optimizing Automatic Deployment Using Non-Functional Requirement Annotations. In: *ISOLA'08: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, October 2008.
Wang2008b	Zhonglei Wang, Wolfgang Haberl, Andreas Herkersdorf, and Martin Wechs: A Simulation Approach for Performance Validation during Embedded Systems Design. In: *ISOLA'08: International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Porto Sani, Greece, October 2008.
Herrmannsdoerfer2009	Markus Herrmannsdoerfer, Wolfgang Haberl, and Uwe Baumgarten: Model-level Simulation for COLA. In: *MISE'09: International Workshop on Modeling in Software Engineering*, Vancouver, Canada, May 2009.
Haberl2009a	Wolfgang Haberl, Stefan Kugele, and Uwe Baumgarten: Reliable Operating Modes for Distributed Embedded Systems. In: *MOMPES'09: International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, Vancouver, Canada, May 2009.

Haberl2009b	Wolfgang Haberl, Michael Tautschnig, and Uwe Baumgarten: Generating Distributed Code from COLA Models. In: *Trends in Communication Technologies and Engineering Science*, pages 265-279, Lecture Notes in Electrical Engineering, Volume 33, Springer Netherlands, May 2009.
Wang2009	Zhonglei Wang, Wolfgang Haberl, Andreas Herkersdorf, and Martin Wechs: SysCOLA – A Framework for Co-Development of Automotive Software and System Platform. In: *DAC'09: Design Automation Conference*, San Francisco, USA, July 2009.
Haberl2010a	Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten: Model-Level Debugging of Embedded Real-Time Systems. In: *CIT'10: IEEE International Conference on Computer and Information Technology*, Bradford, UK, June 2010.
Haberl2010b	Wolfgang Haberl, Stefan Kugele, and Uwe Baumgarten: Model-Based Generation of Fault-Tolerant Embedded Systems. In: *ESA'10: International Conference on Embedded Systems and Applications*, Las Vegas, USA, July 2010.
Haberl2010c	Wolfgang Haberl, Markus Herrmannsdoerfer, Stefan Kugele, Michael Tautschnig, and Martin Wechs: Seamless Model-Driven Development Put into Practice. In: *ISoLA'10: International Symposium on Leveraging Applications*, Heraklion, Greece, October 2010.

Bibliography

[1] P. A. Abdulla, J. Deneux, G. Stålmarck, H. Ågren, and O. Åkerlund. Designing safe, reliable systems using scade. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 4313 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2004.

[2] W. B. Ackerman. Data flow languages. *IEEE Computer*, 15(2):15–25, 1982.

[3] A. Albert. Comparison of event-triggered and time-triggered concepts with regard to distributed control systems. In *Embedded World 2004*, Jan 2004.

[4] N. Amálio, S. Stepney, and F. Polack. Formal proof from uml models. In J. Davies, W. Schulte, and M. Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2004.

[5] C. Ananda. Civil aircraft advanced avionics architecutres - an insight into saras avionics, present and future perspective. 2007.

[6] P. Anderson. Coding standards for high-confidence embedded systems. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1 –7, 2008.

[7] S. Arthur, H. N. Breed, and C. Schmitt-Luehmann. Shifting car makeup shakes up oem status quo: Software strength is critical. Published online, 2003.

[8] AUTOSAR GbR. Layered Software Architecture, V2.2.2 R3.1.

[9] AUTOSAR GbR. Technical Overview, V2.2.2 R3.1.

[10] AUTOSAR GbR. Specification of RTE, V2.3.0 R3.1.

[11] AUTOSAR GbR. Specification of Operating System, V3.1.1 R3.1.

[12] AUTOSAR GbR. Software Component Template, V3.3.0 R3.1.

[13] A. Avritzer, E. J. Weyuker, and C. M. Woodside, editors. *Proceedings of the 7th International Workshop on Software and Performance, WOSP 2008, Princeton, NJ, USA, June 23-26, 2008.* ACM, 2008.

[14] K. Balasubramanian, A. S. Krishna, E. Turkay, J. Balasubramanian, J. Parsons, A. S. Gokhale, and D. C. Schmidt. Applying model-driven development to distributed real-time and embedded avionics systems. *IJES*, 2(3/4):142–155, 2006.

[15] A. Bauer, M. Leucker, and J. Streit. SALT - structured assertion language for temporal logic. In Z. Liu and J. He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 757–775. Springer, 2006.

[16] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270 –1282, Sept. 1991.

[17] A. Benveniste, P. Bournai, T. Gautier, M. Le Borgne, P. Le Guernic, and H. Marchand. The signal declarative synchronous language: controller synthesis and systems/architecture design. In *Decision and Control, 2001. Proceedings of the 40th IEEE Conference on*, volume 4, pages 3284–3289 vol.4, 2001.

[18] A. Benveniste, B. Caillaud, and P. Le Guernic. Compositionality in dataflow synchronous languages: specification & distributed code generation. *Information and Computation*, 163(1):125–171, 2000.

[19] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[20] S. Berger. Arinc 629 digital communication system – application on the 777 and beyond. *Microprocessors and Microsystems*, 20(8):463 – 471, 1997.

[21] K. Berkenkötter. Using UML 2.0 in real-time development – a critical review. In *SVERTS Workshop at the UML 2003 Conference (October 2003)*, Jan 2003.

[22] G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, 1992.

[23] J. Berwanger, C. Ebner, A. Schedl, R. Belschner, S. Fluhrer, P. Lohrmann, E. Fuchs, D. Millinger, M. Sprachmann, F. Bogenberger, G. Hay, A. Krüger, M. Rausch, W. O. Budde, P. Fuhrmann, and R. Mores. Flexray – the communication system for advanced automotive control systems. *SAE transactions*, 110(7):303–314, Jan 2001.

[24] F. Brajou and P. Ricco. The airbus A380 - an AFDX-based flight test computer concept. In *AUTOTESTCON 2004. Proceedings*, pages 460 – 463, 2004.

[25] L. C. Briand and A. L. Wolf, editors. *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, 2007.

[26] M. Broy. Automotive software and systems engineering. In *MEMOCODE*, pages 143–149. IEEE, 2005.

[27] M. Broy. Challenges in automotive software engineering. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 33–42. ACM, 2006.

[28] M. Broy. The 'grand challenge' in informatics: Engineering software-intensive systems. *IEEE Computer*, 39(10):72–80, 2006.

[29] M. Broy, M. Feilkas, J. Grünbauer, A. Gruler, A. Harhurin, J. Hartmann, B. Penzenstadler, B. Schätz, and D. Wild. Umfassendes Architekturmodell für das Engineering eingebetteter software-intensiver Systeme (in German). *Technische Universität München, Tech. Rep. TUM- I*, 6, 2008.

[30] M. Broy, I. Kruger, A. Pretschner, and C. Salzmann. Engineering automotive software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[31] A. Burns. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6(3):116 –128, May 1991.

[32] R. Butler, J. Caldwell, and B. Di Vito. Design strategy for a formally verified reliable computing platform. In *Computer Assurance, 1991. COMPASS '91, Systems Integrity, Software Safety and Process Security. Proceedings of the Sixth Annual Conference on*, pages 125 –133, June 1991.

[33] T. Carpenter, K. Driscoll, K. Hoyme, and J. Carciofini. Arinc 659 scheduling: Problem definition. In *IEEE Real-Time Systems Symposium*, pages 165–169. IEEE Computer Society, 1994.

[34] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From simulink to scade/lustre to tta: a layered approach for distributed embedded applications. In *LCTES*, pages 153–162. ACM, 2003.

[35] A. Cimatti, F. Giunchiglia, P. Pecchiari, B. Pietra, J. Profeta, D. Romano, P. Traverso, and B. Yu. A provably correct embedded verifier for the certification of safety critical software. In O. Grumberg, editor, *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 202–213. Springer, 1997.

[36] A. Cook. Arinc 653 – challenges of the present and future. *Microprocessors and Microsystems*, 19(10):575 – 579, 1995.

[37] P. Cuenot, D.-J. Chen, S. Gérard, H. Lönn, M.-O. Reiser, D. Servat, R. T. Kolagari, M. Törngren, and M. Weber. Towards improving dependability of automotive systems by using the east-adl architecture description language. In R. de Lemos, C. Gacek, and A. B. Romanovsky, editors, *WADS*, volume 4615 of *Lecture Notes in Computer Science*, pages 39–65. Springer, 2006.

[38] J. Dannenberg and C. Kleinhans. The coming age of collaboration in the automotive industry. *Mercer Management Journal*, Jan 2004.

[39] J. DeAntoni, F. Mallet, F. Thomas, G. Reydet, J.-P. Babau, C. Mraidha, L. Gauthier, L. Rioux, and N. Sordon. RT-simex: retro-analysis of execution traces. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 377–378, Santa Fe, New Mexico, USA, 2010. ACM.

[40] V. Debruyne, F. Simonot-Lion, and Y. Trinquet. East-adl — an architecture description language. In P. Dissaux, M. Filali-Amine, P. Michel, and F. Vernadat, editors, *Architecture Description Languages*, volume 176 of *IFIP International Federation for Information Processing*, pages 181–195. Springer Boston, 2005.

[41] S. Demathieu, F. Thomas, C. André, S. Gérard, and F. Terrier. First experiments using the uml profile for marte. In *ISORC*, pages 50–57. IEEE Computer Society, 2008.

[42] F. Dudenhöffer. Elektronik-Ausfälle: Tendenz steigend. *Hanser Automotive Electronics — Systems*, 3–4:12–14, 2004.

[43] C. Ebert and J. Salecker. Guest editors' introduction: Embedded software technologies and trends. *IEEE Software*, 26(3):14–18, 2009.

[44] M. Faugère, T. Bourbeau, R. de Simone, and S. Gérard. MARTE: Also an UML profile for modeling AADL applications. In *ICECCS*, pages 359–364. IEEE Computer Society, 2007.

[45] H. Fecher, J. Schönborn, M. Kyas, and W. P. de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2005.

[46] H. Fennel, L. Lundh, J. Leflour, J. Maté, and K. Nishikawa. Autosar–challenges and achievements 2005. *autosar.de*.

[47] R. B. France, A. Evans, K. Lano, and B. Rumpe. The uml as a formal modeling notation. *Computer Standards & Interfaces*, 19(7):325–334, 1998.

[48] R. B. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In Briand and Wolf [25], pages 37–54.

[49] J. Gait. A probe effect in concurrent programs. *Software, Practice and Experience*, 16(3):225–233, 1986.

[50] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Comput. Surv.*, 31(1):1–26, 1999.

[51] P. Gerum. Xenomai-implementing a rtos emulation framework on gnu/linux. *inf.pucrs.br*, Jan 2004.

[52] K. Grimm. Software technology in an automotive company - major challenges. In *ICSE*, pages 498–505. IEEE Computer Society, 2003.

[53] W. Haberl, J. Birke, and U. Baumgarten. A middleware for model-based embedded systems. In H. R. Arabnia and Y. Mun, editors, *ESA*, pages 253–259. CSREA Press, 2008.

[54] W. Haberl, M. Herrmannsdoerfer, J. Birke, and U. Baumgarten. Model-level debugging of embedded real-time systems. In *CIT*, pages 1887–1894. IEEE Computer Society, 2010.

[55] W. Haberl, M. Herrmannsdoerfer, S. Kugele, M. Tautschnig, and M. Wechs. Seamless model-driven development put into practice. In T. Margaria and B. Steffen, editors, *ISoLA (1)*, volume 6415 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2010.

[56] W. Haberl, S. Kugele, and U. Baumgarten. Reliable operating modes for distributed embedded systems. In *MOMPES*, pages 11–21. IEEE Computer Society, 2009.

[57] W. Haberl, S. Kugele, and U. Baumgarten. Model-based generation of fault-tolerant embedded systems. In H. R. Arabnia and A. M. G. Solo, editors, *ESA*, pages 136–142. CSREA Press, 2010.

[58] W. Haberl, M. Tautschnig, and U. Baumgarten. From cola models to distributed embedded systems code. *IAENG International Journal of Computer Science*, 35(3):427–437, 2008.

[59] W. Haberl, M. Tautschnig, and U. Baumgarten. Running cola on embedded systems. *Proceedings of the 2008 International MultiConference of Engineers and Computer Scientists*, 2008.

[60] N. Halbwachs. Synchronous programming of reactive systems. In A. J. Hu and M. Y. Vardi, editors, *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1998.

[61] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305 –1320, Sept. 1991.

[62] H. Hansson, H. W. Lawson, O. Bridal, C. Eriksson, S. Larsson, H. Lön, and M. Strömberg. Basement: An architecture and methodology for distributed automotive real-time systems. *IEEE Trans. Computers*, 46(9):1016–1027, 1997.

[63] B. Hardung, T. Kölzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In G. C. Buttazzo, editor, *EMSOFT*, pages 203–210. ACM, 2004.

[64] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[65] D. Harel and A. Pnueli. *On the development of reactive systems*, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[66] H. Heinecke, W. Damm, B. Josko, A. Metzner, H. Kopetz, A. L. Sangiovanni-Vincentelli, and M. D. Natale. Software components for reliable automotive systems. In *DATE*, pages 549–554. IEEE, 2008.

[67] C. Heinisch and M. Simons. Adaptierbare Software-Architektur für den Software-Download in Kfz-Steuergeräte (in German). In *GI Jahrestagung (1)*, volume 34 of *LNI*, pages 320–324. GI, 2003.

[68] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[69] T. A. Henzinger, C. M. Kirsch, and S. Matic. Composable code generation for distributed giotto. In Y. Paek and R. Gupta, editors, *LCTES*, pages 21–30. ACM, 2005.

[70] T. A. Henzinger and J. Sifakis. The discipline of embedded systems design. *IEEE Computer*, 40(10):32–40, 2007.

[71] M. Herrmannsdoerfer, W. Haberl, and U. Baumgarten. Model-level simulation for cola. In *Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, MISE '09, pages 38–43, Washington, DC, USA, 2009. IEEE Computer Society.

[72] R. Hierl. Optimierung und erweiterung des codegenerators für die component language (COLA). Diploma thesis, TU München, 2008.

[73] D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[74] S. S. II, J. W. Rozenblit, and K. Buchenrieder. Multilevel testing for design verification of embedded systems. *IEEE Design & Test of Computers*, 19(2):60–69, 2002.

[75] E. K. Jackson, E. Kang, M. Dahlweid, D. Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for mda. In L. P. Carloni and S. Tripakis, editors, *EMSOFT*, pages 39–48. ACM, 2010.

[76] R. Kirner and P. P. Puschner. Classification of code annotations and discussion of compiler-support for worst-case execution time analysis. In R. Wilhelm, editor, *WCET*, volume 1 of *OASICS*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[77] H. Kopetz. Event-triggered versus time-triggered real-time systems. In A. I. Karshmer and J. Nehmer, editors, *Operating Systems of the 90s and Beyond*, volume 563 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 1991.

[78] H. Kopetz. Why time-triggered architectures will succeed in large hard real-time systems. In *FTDCS*, pages 2–9. IEEE Computer Society, 1995.

[79] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Boston, MA, 1997.

[80] H. Kopetz. The time-triggered architecture. In *ISORC*, pages 22–. IEEE Computer Society, 1998.

[81] H. Kopetz. A comparison of ttp/c and flexray. *Research Report*, Jan 2001.

[82] H. Kopetz and G. Grunsteidl. TTP – a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, Jan. 1994.

[83] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Trans. Computers*, 36(8):933–940, 1987.

[84] S. Kugele and W. Haberl. Mapping data-flow dependencies onto distributed embedded systems. In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 272–278. CSREA Press, 2008.

[85] S. Kugele, W. Haberl, M. Tautschnig, and M. Wechs. Optimizing automatic deployment using non-functional requirement annotations. In T. Margaria and B. Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 400–414. Springer, 2008.

[86] S. Kugele, M. Tautschnig, A. Bauer, C. Schallhart, S. Merenda, W. Haberl, C. Kühnel, F. Müller, Z. Wang, D. Wild, S. Rittmann, and M. Wechs. Cola – the component language. Technical Report TUM-I0714, 2007.

[87] C. H. LeDoux and D. S. Parker, Jr. Saving traces for ada debugging. In *Proceedings of the 1985 annual ACM SIGAda international conference on Ada*, SIGAda '85, pages 97–108, New York, NY, USA, 1985. Cambridge University Press.

[88] E. A. Lee. What's ahead for embedded software? *IEEE Computer*, 33(9):18–26, 2000.

[89] G. Leen and D. Heffernan. Expanding automotive electronic systems. *IEEE Computer*, 35(1):88–93, 2002.

[90] P. Liggesmeyer and M. Trapp. Trends in embedded software engineering. *IEEE Software*, 26(3):19–25, 2009.

[91] C. D. Locke. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. *Real-Time Systems*, 4(1):37–53, 1992.

[92] J. Ludwig. Effiziente FAS-Entwicklung. *Hanser Automotive Electronics - Systems*, 9:36–38, 2010.

[93] R. Luo and M. Kay. Multisensor integration and fusion in intelligent systems. *Systems, Man and Cybernetics, IEEE Transactions on*, 19(5):901 –931, 1989.

[94] D. A. Mackall. Development and flight test experiences with a flight-crucial digital control system. Technical report, NASA Dryden Flight Research Center, Edwards, CA, 1988.

[95] R. Maier, G. Bauer, G. Stöger, and S. Poledna. Time-triggered architecture: A consistent computing platform. *IEEE Micro*, 22(4):36–45, 2002.

[96] R. Makowitz and C. Temple. Flexray - a communication network for automotive control systems. In *Factory Communication Systems, 2006 IEEE International Workshop on*, pages 207 –212, 2006.

[97] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Sci. Comput. Program.*, 46(3):219–254, 2003.

[98] P. Marwedel. *Embedded System Design.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[99] S. Matic, M. Goraczko, J. Liu, D. Lymberopoulos, B. Priyantha, and F. Zhao. Resource modeling and scheduling for extensible embedded platforms. Technical Report MSR-TR-2006-176, 2006.

[100] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys (CSUR)*, 21(4):593–622, 1989.

[101] G. C. Meijer. Concepts and focus point for intelligent sensor systems. *Sensors and Actuators A: Physical*, 41(1-3):183–191, 1994.

[102] MIRA Ltd. *MISRA-C: Guidelines for the Use of the C Language in Critical Systems*, October 2004.

[103] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert. Trends in automotive communication systems. *Proceedings of the IEEE*, 93(6):1204–1223, 2005.

[104] G. C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–94, 2000.

[105] OSEK/VDX Group. *OIL: OSEK Implementation Language, Version 2.5*, 2004.

[106] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[107] P. Peti, R. Obermaisser, F. Tagliabo, A. Marino, and S. Cerchio. An integrated architecture for future car generations. In *ISORC*, pages 2–13. IEEE Computer Society, 2005.

[108] A. Pretschner, M. Broy, I. H. Krüger, and T. Stauner. Software engineering for automotive systems: A roadmap. In Briand and Wolf [25], pages 55–71.

[109] P. Prisaznuk. Integrated modular avionics. In *Aerospace and Electronics Conference, 1992. NAECON 1992., Proceedings of the IEEE 1992 National*, pages 39 –45 vol.1, May 1992.

[110] P. Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1.E.5–1 –1.E.5–10, 2008.

[111] S. Rittmann. *A methodology for modeling usage behavior of multi-functional systems*. PhD thesis, 2008.

[112] Robert Bosch GmbH. *Kraftfahrtechnisches Taschenbuch*. 2007.

[113] L. Rodrigues, M. L. Guimarães, and J. Rufino. Fault-tolerant clock synchronization in CAN. In *IEEE Real-Time Systems Symposium*, pages 420–429, 1998.

[114] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[115] J. M. Rushby. Bus architectures for safety-critical embedded systems. pages 306–323, 2001.

[116] A. L. Sangiovanni-Vincentelli and M. D. Natale. Embedded system design for automotive applications. *IEEE Computer*, 40(10):42–51, 2007.

[117] A. Silberschatz, P. B. Galvin, and G. Gagne. Operating system concepts. Jan 2009.

[118] M. Stringfellow, N. Leveson, and B. Owens. Safety-driven design for software-intensive aerospace and automotive systems. *Proceedings of the IEEE*, 98(4):515 –525, 2010.

[119] J. Swingler and J. McBride. The degradation of road tested automotive connectors. In *Electrical Contacts, 1999., Proceedings of the Forty-Fifth IEEE Holm Conference on*, pages 146 –152, 1999.

[120] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 403–408. Thomson Book Company, Washington D.C., 1968.

[121] K. Tindell, H. Hansson, and A. Wellings. Analysing real-time communications: controller area network (CAN). In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 259 –263, 1994.

[122] K. Tindell, H. Kopetz, F. Wolf, and R. Ernst. Safe automotive software development. In *DATE*, pages 10616–10623. IEEE Computer Society, 2003.

[123] R. van Engelen, D. B. Whalley, and X. Yuan. Validation of code-improving transformations for embedded systems. In *SAC*, pages 684–691. ACM, 2003.

[124] R. van Engelen, D. B. Whalley, and X. Yuan. Automatic validation of code-improving transformations on low-level program representations. *Sci. Comput. Program.*, 52:257–280, 2004.

[125] K. Venkatesh Prasad, M. Broy, and I. Krueger. Scanning advances in aerospace & automobile software technology. *Proceedings of the IEEE*, 98(4):510–514, 2010.

[126] S. Voget. Autosar and the automotive tool chain. In *DATE*, pages 259–262. IEEE, 2010.

[127] S. Wagner, B. Schätz, S. Puchner, and P. Kock. A case study on safety cases in the automotive domain: Modules, patterns, and models. In *ISSRE*, pages 269–278. IEEE Computer Society, 2010.

[128] H. Wallentowitz and K. Reif. Handbuch Kraftfahrzeugelektronik: Grundlagen, Komponenten, Systeme, Anwendungen (in German). *Vieweg, Wiesbaden*, Jan 2006.

[129] Z. Wang, W. Haberl, S. Kugele, and M. Tautschnig. Automatic generation of SystemC models from component-based designs for early design validation and performance analysis. In Avritzer et al. [13], pages 139–144.

[130] Z. Wang, A. Sanchez, and A. Herkersdorf. SciSim: a software performance estimation framework using source code instrumentation. In Avritzer et al. [13], pages 33–42.

[131] D. Wybo and D. Putti. A qualitative analysis of automatic code generation tools for automotive powertrain applications. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 225 –230, 1999.

[132] J. Xu and D. L. Parnas. Scheduling processes with release times, deadlines, precedence, and exclusion relations. *IEEE Trans. Software Eng.*, 16(3):360–369, 1990.

[133] A. Zahir. Oil – osek implementation language. In *OSEK/VDX Open Systems in Automotive Networks (Ref. No. 1998/523), IEEE Seminar*, pages 8/1 –8/3, Nov. 1998.

[134] W. Zheng, Q. Zhu, M. D. Natale, and A. L. Sangiovanni-Vincentelli. Definition of task allocation and priority assignment in hard real-time distributed systems. In *RTSS*, pages 161–170. IEEE Computer Society, 2007.