# Applying Model-Driven Design and Development to Distributed Time-Triggered Systems

Yu Sun
Computer and Information Sciences, University of Alabama at Birmingham,
Birmingham, AL, USA
yusun@cis.uab.edu

and

Christoph Wienands, Meik Felser
GTF System Architecture and Platforms, Corporate Research, Siemens Corporation
Princeton, NJ, USA
{christoph.wienands, meik.felser}@siemens.com

**ABSTRACT**

This paper describes the experiences with a model-driven approach to design and create a time-triggered system based on FlexRay. The development and maintenance of such systems typically requires much error-prone and detailed work, such as calculating transmission and execution schedules, protocol initialization and configuration, and data transmit and receive configurations. A graphical domain-specific language allows for efficiently modeling physical and logical aspects of the system as well as quality of service requirements (QoS) by providing an abstraction over said details. A model completion step using scheduling and arbitration algorithms then derives missing information. A code generation step creates target platform- and controller-specific code from the model.

**Keywords:** Domain-specific language, time- triggered system, FlexRay, Fibex, scheduling.

## 1. INTRODUCTION

Time-triggered systems have fixed communication and execution schedules based on a common clock source, which allows for predictable and deterministic behavior of such systems. This makes them excellent target architectures for real-time systems [18]. For example, time-triggered systems are widely used in the automobile industry. On the other hand, large time-triggered systems require even larger amounts of information points, for scheduling function execution and signal transmission, protocol initialization, and configuration of controllers. At the same time these information points are highly dependent on each other, which makes efficient, manual development pretty much impossible.

Our goal was to develop a universal, model-driven approach for the development of time-triggered system. Developers would focus on 1) physical aspects such as electronic control units (ECUs), controllers, wiring, 2) logical aspects such as functions, ports, signals, 3) deployment spects, the assignment of functions to ECUs, and 4) quality attributes such as maximum signal latency and reliability. This approach was realized through a graphical domain-specific modeling language (DSL) and a code generation tool chain. Until now, it was tested and evaluated in a simulator setup using multiple participating ECUs.

## 2. BACKGROUND AND TECHNOLOGIES

### Time-triggered systems

A real-time system has to perform certain tasks within a specified amount of time to meet deadlines dictated by its environment [18]. In time-triggered real-time systems this requirement is fulfilled by periodically initiating activities at predetermined points in time. Time-triggered systems are widely used in safety-critical applications due to their overall predictability.

### Communication Buses and Models

**FlexRay**: The basis of the distributed time-triggered system we designed and developed is the FlexRay bus [1][19]. FlexRay is a vehicle bus designed to overcome the limitations of the CAN bus used in automobiles for communication between the ECUs.

FlexRay uses a TDMA (Time Division Multiple Access) access scheme, where each node waits for its turn to write on the bus. For this, the FlexRay specification includes a time synchronization protocol that establishes a shared time base between the distributed nodes. Communication on the bus is organized in communication cycles, which can contain different types of data. In our example we concentrate on static segments. They are divided into slots and each slot can contain a reserved frame of data that is assigned to a specific ECU. The ECU has the opportunity to transmit data when the slot occurs in time. If the slot is missed, the node has to wait for the next

communication cycle to write and transmit the data in that slot. Due to this fixed scheduling scheme, FlexRay can guarantee a deterministic delivery of messages, as well as overall data throughput.

Additionally FlexRay usually comes with two channels (physical wire pairs) that can be used independently but more often are used for redundant message transfer to provide a fault-tolerant communication system.

In contrast to event-based communication systems such as CAN, each node must know the exact communication schedule and needs to be programmed with the correct network parameters before it can participate on the bus. Consistency in programming the nodes can be achieved by using a network description model such as FIBEX.

**FIBEX**: FIBEX (Field Bus Exchange Format) [2] is an XML-based format to represent the configuration of message-oriented communications systems. It was developed and standardized by ASAM (Association for Standardization of Automation and Measuring Systems) especially for use in vehicle networks.

FIBEX provides the facility to describe the information needed to design a network cluster of ECUs on all levels down to the bit level. Network parameters saved in FIBEX provides all participants in the engineering process (such as network designers, prototypers, validators, testers) the ability to share and exchange the network configurations and configure test tools, simulators or ECUs. FIBEX is not limited to describe FlexRay-based networks but currently also supports other communication bus technologies such as CAN, MOST and LIN.

We used FIBEX as the base of our derived network description model.

### Modeling Tool Chain

The complete set of applied modeling technologies is described in Table 1. The most influential technological choice was the use of Graphical Modeling Framework (GMF) [20]. It was chosen because it is one of the most powerful, freely available DSL workbenches. The Epsilon and Eugenia tool chains were chosen based on previous experience with the complexities and inefficiencies of GMF [13].

TABLE I. MODELING TOOLS

| Technology | Description |
|---|---|
| Eclipse Modeling Framework (EMF) | Modeling- and meta-modeling libraries. |
| Graphical Modeling Framework (GMF) [20] | Eclipse-based workbench for graphical DSLs. |
| Java Emitter Templates (JET) | Both used for customization of EMF and GMF code generation templates. |
| openArchitectureWare Xpand | |
| Epsilon Object Language (EOL) [7] | Powerful scripting language, basis for Epsilon model validation and model transformation languages. |
| Epsilon Eugenia [10] | GMF add-on for efficient, repeatable DSL generation from a single model. |

## 3. OVERVIEW OF THE SOLUTION

To handle the complexity of the tasks involved in building a distributed time-triggered software system, we propose a model-driven solution to raise the level of abstractions so that a separation of the time-triggered domain problem from the underlying implementation details could be realized, leading to an alleviation of the complexity.

The solution is based on the principle of Model-Driven Engineering (MDE) [3] and Domain-Specific Modeling (DSM) [4], which decouples the description of the essential characteristics of a problem from the details of a specific solution space (e.g., middleware, programming languages). This is accomplished through a domain-specific modeling language (DSL) [5], here called Time-Triggered Systems Modeling Language or TTS-ML, to declaratively define a software system using high-level domain concepts, and automatically generate the desired low-level software artifacts (e.g., programming code, simulation script, XML deployment description) by model transformation engines and generators.

Figure 1 is an overview of the solution. The TTS-ML is defined as a platform-independent metamodel, which enables users to specify the high-level distributed time-triggered system model instances. The high-level model only contains the direct concepts and specification about the problem domain, without any low-level scheduling or communication information, as well without any implementation details. From the high-level model, a model completion step generates a complete model automatically through model-to-model transformation [6], calculating and incorporating all the necessary low-level information to the model. Finally, various types of platform-dependent implementations (i.e., codes) can be generated from the models using several code generators.

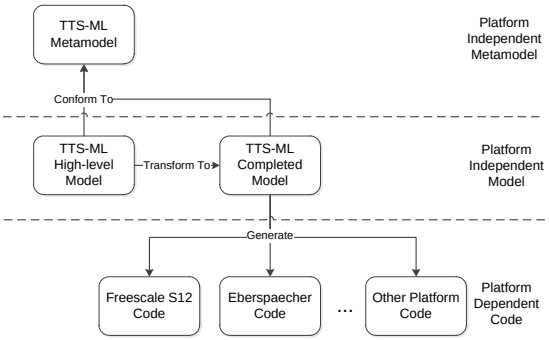


Figure 1. Overview of the Solution

Using this solution, users can specify their desired distributed time-triggered systems in a domain-specific way, which not only improves the efficiency, accuracy, and reliability of the system development, but potentially enables domain experts who are not experienced in software development to participate in the system design and implementation. The details of the solution will be given in the following sections.

## 4. TIME-TRIGGERED SYSTEMS MODELING LANGUAGE

The TTS-ML's metamodel was derived from the FIBEX specification [2]. TTS-ML then added a number of modeling concepts, such as function execution schedule elements (FIBEX does not cover function execution schedules) and quality attribute properties.

The graphical modeling language can be used to specify three aspects of the system: 1) the physical view of the system hardware architecture 2) the logical view of the data flows among functions 3) the deployment view of the function assignment. These views were, because of their intuitiveness, derived from the 4+1 view model for software architectures [21].

### Physical View - System Architecture

The physical view specifies how the hardware components are configured and connected within the system. Figure 2 shows an example of specifying the physical view in the high-level model.

The cluster is made up of all ECUs (Electronic Control Units) linked to one physical bus, and therefore share the same communication medium and protocol, which for TTS-ML is FlexRay. Channels (wire pairs) connect different ECUs. The FlexRay bus specifies two channels in order to create a more reliable communication link. Each ECU contains a FlexRay controller that is responsible for sending and receiving data frames from the communication medium.

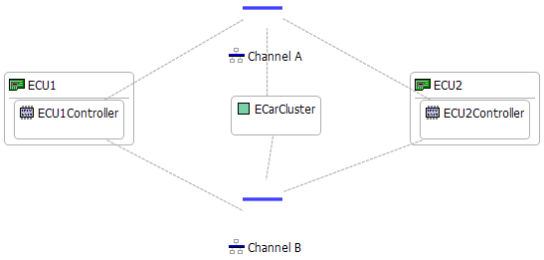


Figure 2. An example of modeling the physical view

Some cluster-wide attributes describing the communication protocol are *pKeySlotId* and *pWakeUpChannel*; some local properties about the timing can be defined per controller (e.g., *MicroPerCycle*, *RateCorrectionOut*, etc.) A complete physical model represents an ensemble of ECUs connected correctly and ready to work.

### Logical View – Function Data Flows

Functions are executed on different ECUs. It is very common that functions need to communicate with each other to collaborate on some of the tasks. Data communication among functions involves data flows and signals. Therefore, the main purpose of the logical view is to specify the functions, input and output signals of functions and the correspondent data flows that connect function input ports and output ports.

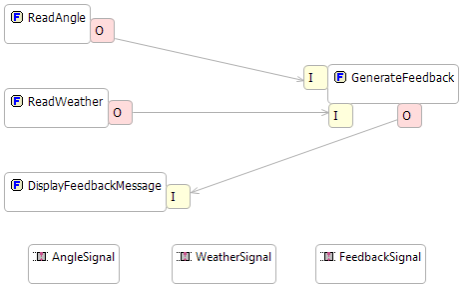


Figure 3. An example of modeling the logical view

As shown in Figure 3, functions have input and output ports. Dataflows connect an output port to one or more input ports. A signal exist in each set of connected ports, which provides the information about the data to be communicated between the ports, such as the data type or the data length.

### Deployment View - Function Assignment

While the logical view specifies the data communication among functions, the deployment view assigns each function to an ECU simply by making connections between a function and an ECU.

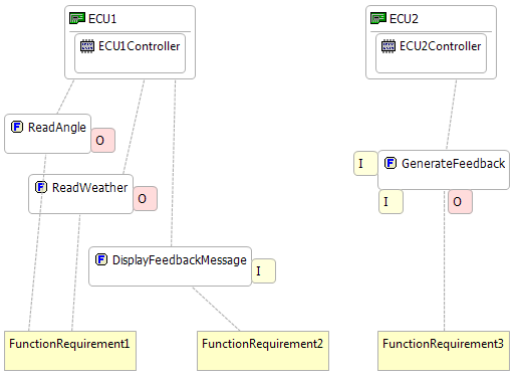


Figure 4. An example of deployment view

Another important aspect is the definition of function execution requirements for each function to give timing constraints on its execution. The most essential execution requirement attributes are: *Cycle Period* (i.e., the inverse of the frequency of function execution), *Worst Case Execution Time (WCET)* (i.e., the duration of executing a function in the worst case), and *Cycle Offset* (i.e., the first cycle to execute the function). These attributes are the fundamental elements for calculating function execution and signal communication schedules in the model completion step.

## 5. MODEL COMPLETION

The high-level model represented by the three views only specifies the static architecture and function configuration. In order to generate the implementation code, more details about the dynamic execution and transmission information are necessary, such as when and how often to execute each function, when and how often to transmit and receive data in each ECU, how to package the signal data and how to dispatch it. Although the

corresponding elements (e.g., *FrameTriggering*, *FunctionExecutionMatrix*) have been defined in the metamodel of TTS-ML (inherited from FIBEX), it is a complex, tedious and error-prone task to specify this information manually, particularly when a huge number of functions and data flow connections exists in the system. Thus, a model completion process is implemented to automatically generate the scheduling matrices and complete the high-level model with low-level information. This completed model serves as the basis for code generation.

The model completion consists of the flowing steps:

**Step 1 – Model validation**

Before model completion, it must be ensured that all the necessary model elements and properties have already been specified in the high-level model instance correctly. For instance, each function must be assigned to an ECU; each function must have a function requirement with all the timing constraints; the signal must contain the specific data information; and more. Missing any of these will lead to failures when calculating the scheduling matrices and adding extra low-level model elements. A set of rules have been defined in Epsilon Validation Language (EVL) [7] to check if the model satisfies all the constraints. Figure 5 is an example of the rule, which means that the Cycle Period specified for a certain function as a timing constraint must be a non-zero value.

```
context FunctionReqType {
   constraint periodNotZero {
        check: self.CyclePeriod <> 0.0
        message: 'FunctionReqType '
                 + self.Id + ' must have
                 a non-zero CyclePeriod'
   }
}
```

Figure 5. An example of a model checking rule

**Step 2 – Generate function execution schedule matrix.**

The correct execution of the whole system is realized by the execution of functions assigned to ECUs. Once all functions are assigned and the function execution requirement for each function is specified in the high-level model, a function execution matrix can be generated for each ECU through the following simple algorithm:

1. Calculate the duration of a single FlexRay cycle.
2. Convert the Cycle Period, Cycle Offset, WCET of each function requirement from seconds to cycles.
3. For each ECU, calculate the total number of cycles in a schedule (called *cycle block*) using the least common multiple of all the function Cycle Period.
4. For each function assigned to an ECU, try to place its execution blocks according to its period from the very first available cycle in the cycle block. If a certain cycle is not available (i.e., the cycle is already assigned to other functions), try to shift the function's schedule by one cycle.
5. If all the functions in an ECU are assigned to the right number of execution cycles, the function execution scheduling succeeds; as of now, if a function cannot be assigned enough cycles, scheduling fails.

If the function execution scheduling succeeds on an ECU, a list of function execution entries will be added to the function execution matrix of that ECU, specifying all the cycles that are used and what function will be executed on those cycles. The matrix data is integrated into the model trough a model-to-model transformation implemented using Epsilon Transformation Language (ETL) [7].

A function scheduling matrix visualization is also implemented to demonstrate the scheduling as shown in Figure 6.

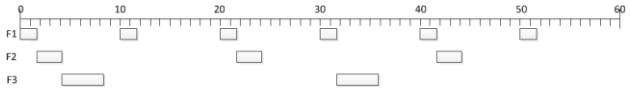


Figure 6. The visualization of function execution schedule

**Step 3 – Generate signal communication schedule matrix.**

The execution of functions requires the correct input signal to produce the right output signal. Therefore, the data at the input port of the function should be received and ready to be used before the execution of the function, and the function must transmit the data from the output port timely for other functions to use.

The generated signal communication schedule uses the static segment in the FlexRay communication block. 64 static slots are available in each FlexRay cycle to transmit / receive data frames. A FlexRay cycle block can contain up to 1023 cycles. The following algorithm is applied to generate the scheduling:

1. Infer a port requirement for each input / output port from the owning function's requirement. The Cycle Period and Cycle Offset of the port requirement should be equal to its function. If the two connected input and output ports have different Cycle Period, the shorter period will override the longer period.
2. Decide the FlexRay cycle(s) in a FlexRay cycle block to transmit / receive the signals according to Cycle Period and Cycle Offset. If a certain cycle is not available (i.e., all the static slots are allocated to other signals in this cycle), shift to the next available cycle.
3. Once the cycle is set, assign static slot(s) based on the length of the data from the first available static slot in this FlexRay cycle.
4. If all the signals can be assigned with enough and correct FlexRay cycles and slots, the scheduling succeeds; if certain signals cannot be assigned with enough cycles and slots with the right Cycle Period, scheduling fails.

All the signals in the same static slot of a FlexRay cycle compose a FlexRay frame, which is the minimum communication unit in FlexRay. Figure 7 shows the cycle-based signal transmission schedule, similar to the function execution schedule.

Once the scheduling succeeds and signal compositions are determined, again all information is integrated into model using model-to-model transformation.

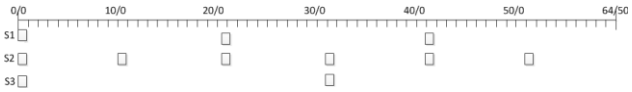


Figure 7. The visualization of signal communication schedule

**Notes about scheduling**

The scheduling algorithm used in this project is a straight-forward fitting algorithm without complex optimization. A number of works have been done on the FlexRay static segment scheduling and optimization [14][15][16][17]. In the future, these algorithms could easily be integrated into the current framework. Besides, after the model completion process, users still can manually modify the model for specific purposes. An *isGenerated* flag is enabled to prevent user-modified elements from being overwritten by the next model completion process.

## 6. CODE GENERATION

The completed model is a platform-independent representation of the whole system. Since there exist various FlexRay controllers placed on various hardware platforms, such as Freescale S12 Microcontroller [8] or Eberspächer FlexRayCard [9], platform-dependent code is needed to be generated to implement the actual system.

In this framework, a code generator is implemented for each type of platform, using Model-To-Text (M2T) transformation [11]. The M2T transformation is specified by a code generation template, which maps each model element to the concrete textual code. A code generation template excerpt is shown in Figure 8, meaning that for each output function port, a variable declaration statement is generated to be used as the buffer for data transmission.

```
«FOREACH getTxSlots(this, ecu) AS output-»
   UINT16 tx_data_«output»
   [«getCluster().PayloadLengthStatic»] = {0};
«ENDFOREACH-»
```

Figure 8. An excerpt of code generation template

A good code generation practice is to design a domain framework, which contains common functionality so that a minimum amount of code needs to be generated from the models [12]. With TTS-ML, a domain framework has been built to handle basic FlexRay protocol operation procedures and the basic hardware configuration. The generated code realizes the FlexRay hardware configuration, function execution initialization and scheduling, signal communication scheduling, signal dispatch and composition, as shown in Figure 9. A good separation of the generated code from the code framework not only reduces the complexity of M2T specifications, but also makes the architecture more maintainable.

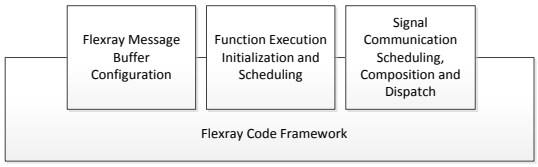


Figure 9. Code framework and generated code

Currently, two generators have been implemented for both the Freescale S12 Microcontroller and Eberspächer FlexRayCard platforms. Other generators can be defined in the same way as extensions.

## 7. TEST AND EVALUATION

To evaluate the framework, a vehicle simulation environment has been setup as illustrated in Figure 10. The environment consists of two Freescale S12 boards and one Eberspächer FlexRayCard connected to a PC. Board B1 connects with a steering wheel, a brake pedal and an acceleration pedal, and provides driver input signals. A car simulation program is running in the PC, which generates the real-time environmental sensor data (e.g., speed, turning angle). A driving assistant algorithm is running in B2, with the function being that if dangerous driving conditions are encountered, direct control of the car by the driver is (partially) overridden with stabilizing behavior by the assistant algorithm in order to automatically reach a safe driving state.

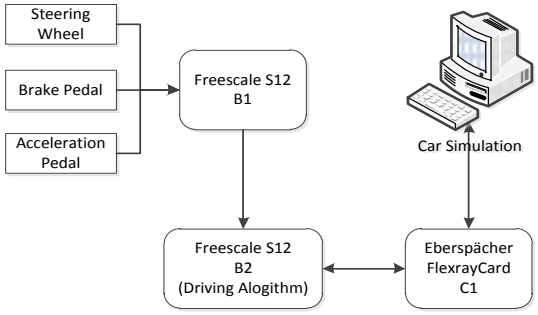


Figure 10. The car simulation environment

The high-level model consists of three views: 1) the physical view with one cluster, three ECUs with three controllers, and two channels, 2) the logical view with six functions and seventy-four signals, 3) the deployment view with six function assignments and four function requirements given to the six functions. After the model completion and code generation steps, individual implementations are generated for the two boards and one card.

## 8. EXPERIENCES

A number of experiences have been gained in the process of designing and implementing the whole framework.

While we did not collect metrics related to effort of DSL and efficiency (see [13] for data on that), this project clearly demonstrated how quality can be raised by using a model-driven approach. Even the small, distributed, time-

triggered system presented in this paper involves a host of hardware configurations, function and signal scheduling information. Manual implementation of the now generated code would have been tedious, time-consuming and error-prone. Additionally as such systems evolve, much of the previous scheduling and configuration effort would have to be repeated. By contrast, creating and evolving a high-level model using appropriate abstractions is much more efficient and results in less chance for errors.

Using the Eugenia tool [10] considerably sped up the creation of the graphical DSL editors. Building and maintaining a modeling tool with GMF is by no means an easy task, which requires six individual models that are highly dependent on each other and all need to be in sync with each other. The Eugenia tool essentially reduces development and maintenance effort down to one model plus some optional, separate customization information.

The FIBEX specification contains a number of good abstractions over the low-level implementation elements and properties. The derivation of TTS-ML from the FIBEX XML schemas had several additional advantages. 1) Domain experts familiar with FIBEX will quickly be able to use TTS-ML even though some additional elements need to be mastered. 2) Model exchange with FIBEX-aware tools and hardware can be accomplished with relative ease. 3) The biggest benefit and time savings, however, came from reusing over one hundred metamodel classes and even more properties that otherwise would have had to be created from scratch. Additionally, FIBEX is compatible with not only FlexRay, but also CAN, MOST, and other protocols, so it provides a good potential to extend the DSL to support other types of distributed systems.

And finally, a welcome side effect is the improvement of overall target system architecture. The design of the domain framework underlying the generated code required the identification of common and variable parts of such systems. By separating those parts and designing changeable and extensible software components, the architecture became more extensible and maintainable, too.

## 9. CONCLUSION AND FUTURE WORK

This paper presents a model-driven approach to develop FlexRay-based, time-triggered, distributed systems. Model completion and code generation were applied to derive platform-dependent models and ultimately platform-specific code from a partial, platform-independent model. The test system could be built in less time and with higher quality. Additionally, the effort of creating the modeling tool chain was significantly reduced compared to previous projects.

Future work will include the development of larger systems with TTS-ML, scheduling tolerance requirements and improved scheduling algorithms, and the possibility to integrate with legacy ECUs that rely on unchangeable, predefined signal transmission schedules.

## REFERENCES


[1] FlexRay Consortium, **FlexRay Communications System, Protocol Specification**, Version 2.1, Revision A, 2005.

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

[3] Douglas C. Schmidt, **Model-Driven Engineering**, IEEE Computer, vol. 39, no. 2, 2006, pages 25-32.

[4] Ákos Lédeczi, Árpád Bakay, Miklós Maróti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, Gábor Karsai, **Composing Domain-Specific Design Environments**, IEEE Computer, vol.34, no. 11, 2001, pages 44-51.

[5] Jeff Gray, Juha-Pekka Tolvanen, Steven Kelly, Aniruddha Gokhale, Sandeep Neema, Jonathan Sprinkle, **Domain-Specific Modeling**, Handbook of Dynamic System Modeling, CRC Press, 2007, Chapter 7, pages 7-1 through 7-20.

[6] Shane Sendall, Wojtek Kozaczynski, **Model Transformation - The Heart and Soul of Model-Driven Software Development**, IEEE Software, vol. 20, no. 5, 2003, pages 42-45.

[7] Eclipse Eplison. http://www.eclipse.org/gmt/epsilon/

[8] Freescale S12 Microcontroller. http://www.freescale.com/

[9] Eberspächer Flexray Card. http://www.eberspacher.com/

[10] Eugenia. http://www.eclipse.org/gmt/epsilon/doc/eugenia/

[11] Manoli Albert, Javier Muñoz, Vicente Pelechano, Oscar Pastor, **Model to Text Transformation in Practice: Generating Code From Rich Associations Specifications**, The 2nd International Workshop on Best Practices in UML (BP-UML06), Tucson, USA, 2006.

[12] Steven Kelly, Juha-Pekka Tolvanen, **Domain-Specific Modeling: Enabling Full Code Generation**, Wiley-IEEE Computer Society Press, 2008.

[13] Christoph Wienands, Michael Golm, **Anatomy of a Visual Domain-Specific Language Project in an Industrial Context**, International Conference on Model Driven Engineering Languages and Systems, Denver, CO, October 2009, pages 453-467.

[14] Haibo Zeng, Wei Zheng, Marco Di Natale, Arkadeb Ghosal, Paolo Giusto, Alberto Sangiovanni-Vincentelli, **Scheduling the FlexRay Bus Using Optimization Techniques**, The 46th IEEE ACM Design Automation Conference, San Francisco, CA, July 2009, pages 874-877.

[15] Martin Lukasiewycz, Michael Glaß, Jürgen Teich, Jürgen Teich, **FlexRay Schedule Optimization of the Static Segment**, In CODE + ISSS, 2009.

[16] Shan Ding, Naohiko Murakami, Hiroyuki Tomiyama, Hiroaki Takada, **A GA-based scheduling method for FlexRay systems**, The 5th ACM International Conference on Embedded Software, September 18-22, 2005, Jersey City, NJ, USA

[17] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, **Bus access optimisation for FlexRay-based distributed embedded systems**, The Conference on Design, Automation and Test in Europe, April 16-20, 2007, Nice, France.

[18] H. Kopetz. **Event-triggered versus time-triggered real-time systems**, Proceedings of the International Workshop on Operating Systems of the 90s and Beyond, LNCS vol. 563, Springer-Verlag, London, UK, 1991, pp. 87–101.

[19] Mathias Rausch, **FlexRay Grundlagen Funktionsweise Anwendung**, 1st ed., Hanser, Munich, Germany, 2008

[20] Graphical Modeling Framework. http://www.eclipse.org/gmf/

[21] Phillippe Kruchten, **Architectural Blueprints – The 4+1 View Model of Software Architecture**, IEEE Software vol.12, no. 6, 1995, pp. 42-50