Carl von
Ossietzky Universität Oldenburg

Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Globally Accurate Locally Inaccurate (GALI): On the Combination of Time-Triggered Architectures with Instruction Accurate Simulators for the Analysis of System Behavior

Von der Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaftender Carl von Ossietzky Universität Oldenburgzur Erlangung des Grades und Titels eines

Doktors der Ingenieurwissenschaften (Dr.-Ing.)

angenommene Dissertation

von Herrn Razi Seyyedi

Razi Seyyedi: *Globally Accurate Locally Inaccurate (GALI):
On the Combination of Time-Triggered Architectures
with Instruction Accurate Simulators
for the Analysis of System Behavior*

Gutachter:
Prof. Dr.-Ing. Wolfgang Nebel

Weitere Gutachter:
Prof. Dr. Martin Fränzle

Tag der Disputation:
2023-03-02

Abstract

Time-Triggered (TT) architectures are widely used in safety-critical computer systems. Time-triggered systems execute tasks based on a predetermined static or dynamic scheduling. Time-triggered system behavior is deterministic, which makes the verification of real-time systems more convenient. There is a demand to test the functionality (behavior) of the integration of complex time-triggered MPSoCs. The functionality is validated together with the system's static timing/schedule concerning the timing behavior of the environment. Existing solutions support high-level functional and low-level target platform implementation models that usually lack observability and debuggability.

In this work, we introduce a novel simulation technique called GALI which stands for "Globally Accurate, Locally Inaccurate" to speed up the simulation of time-triggered systems and its necessary ecosystem targeting more observability and debugging capabilities. The proposed approach consists of a new simulation model based on an instruction-accurate simulation with a predetermined time-triggered system configuration. The simulation technique benefits from the discrete timed execution of a time-triggered architecture and applies it to the instruction accurate simulation and scheduling structure to produce a fast and accurate result. The supporting tooling ecosystem supports the generation of the platform and applications assigned to different processors and partitions for the provided schedule from the user. It generates the board-support package, binary files for different processor architecture, channels for communication, and the corresponding infrastructure required by the time-triggered architecture. The evaluation of this work on a safety-critical and time-sensitive flight control system demonstrates that our simulation technique achieves precisely the same control behavior as the cycle-accurate platform and only has negligible overhead compared to an untimed instruction accurate simulation.

Zusammenfassung

Time-Triggered (TT)-Architekturen sind in sicherheitskritischen Computersystemen weit verbreitet. Zeitgesteuerte Systeme führen Aufgaben basierend auf einer vorbestimmten statischen oder dynamischen Planung aus. Das zeitgesteuerte Systemverhalten ist deterministisch, was die Überprüfung von Echtzeitsystemen vereinfacht. Es besteht der Bedarf, die Funktionalität (Verhalten) der Integration komplexer zeitgetriggerter MPSoCs zu testen. Die Funktionalität wird zusammen mit dem statischen Timing/Zeitplan des Systems in Bezug auf das Timing-Verhalten der Umgebung validiert. Vorhandene Lösungen unterstützen Funktionsmodelle auf hoher Ebene und Implementierungsmodelle auf niedriger Ebene der Zielplattform, denen es normalerweise an Beobachtbarkeit und Debugging-Fähigkeit mangelt.

In dieser Arbeit stellen wir eine neuartige Simulationstechnik namens GALI vor, die für "Globally Accurate, Locally Inaccurate" steht, um die Simulation zeitgesteuerter Systeme und deren notwendigem Ökosystem zu beschleunigen sowie die Beobachtbarkeit und die Debugging-Fähigkeiten zu verbessern. Der vorgeschlagene Ansatz besteht aus einem neuen Simulationsmodell, das auf einer befehlsgenauen Simulation mit einer vorgegebenen zeitgesteuerten Systemkonfiguration basiert. Die Simulationstechnik profitiert von der diskreten zeitgesteuerten Ausführung einer zeitgesteuerten Architektur und wendet sie auf die befehlsgenaue Simulations- und Planungsstruktur an, um ein schnelles und genaues Simulationsergebnis zu erzielen. Das begleitende Tooling-Ökosystem unterstützt die Generierung der Plattform und der Anwendungen, die den verschiedenen Prozessoren und Partitionen unter Berücksichtigung des vom Benutzer vorgegebenen Zeitplans zugewiesen werden. Es generiert das Board-Support-Package, Binärdateien für verschiedene Prozessorarchitekturen, Kommunikationskanäle und die entsprechende Infrastruktur, die von der zeitgesteuerten Architektur benötigt wird. Die Evaluation dieser Arbeit an einem sicherheitskritischen und zeitkritischen Flugsteuerungssystem zeigt, dass unsere Simulationstechnik genau das gleiche Steuerungsverhalten wie die zyklusgenaue Plattform erreicht und nur einen vernachlässigbaren Overhead im Vergleich zu einer nicht zeitgenauen, aber befehlsgenauen Simulation hat.

Publications

Some ideas and figures have appeared previously in the following publications:

[1] Razi Seyyedi, Sören Schreiner, Maher Fakih, Kim Grüttner, and Wolfgang Nebel. "Functional Test Environment for Time-Triggered Control Systems in Complex MPSoCs". In: *Microprocessors and Microsystems* (2020), p. 103072.

[2] Maher Fakih, Kim Grüttner, Sören Schreiner, Razi Seyyedi, Mikel Azkarate-Askasua, Peio Onaindia, Tomaso Poggi, Nera González Romero, Elena Quesada Gonzalez, Timmy Sundström, et al. "Experimental evaluation of SAFEPOWER architecture for safe and power-efficient mixed-criticality systems". In: *Journal of Low Power Electronics and Applications* 9.1 (2019), p. 12.

[3] Razi Seyyedi, Sören Schreiner, Maher Fakih, Kim Grüttner, and Wolfgang Nebel. "Functional Test Environment for Time-Triggered Control Systems in Complex MPSoCs using GALI". In: *2018 21st Euromicro Conference on Digital System Design (DSD)*. IEEE. 2018, pp. 711–718.

[4] Sören Schreiner, Razi Seyyedi, Maher Fakih, Kim Grüttner, and Wolfgang Nebel. "Towards power management verification of time-triggered systems using virtual platforms". In: *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*. 2018, pp. 81–88.

[5] Maher Fakih, Alina Lenz, Mikel Azkarate-Askasua, Javier Coronel, Alfons Crespo, Simon Davidmann, Juan Carlos Diaz Garcia, Nera González Romero, Kim Grüttner, Sören Schreiner, et al. "SAFEPOWER project: Architecture for safe and power-efficient mixed-criticality systems". In: *Microprocessors and Microsystems* 52 (2017), pp. 89–105.

[6] Razi Seyyedi, Mt Mohammadat, Maher Fakih, Kim Grüttner, Johnny Oberg, and Duncan Graham. "Towards virtual prototyping of synchronous real-time systems on NoC-based MPSoCs". In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2017, pp. 1–4.

Contents

Introduction 1

Embedded systems more and more surround us. These systems gain control over so many aspects of our life that the number of embedded systems we have to regard as safety-critical is ever rising. These devices are available, among others, in cars, medical devices, and airplanes, from infrastructure to medicine and transportation.

Such systems usually have a real-time response because of their direct interaction with the environment. The essential characteristic of the real-time system is deterministic processing, which means that we must guarantee that a particular activity concludes within the time boundary or at precise intervals. If the implementation does not match the specification, this application can render a security threat.

A widely used architecture in safety-critical and mixed-criticality computer systems is Time-Triggered Architecture (TTA). These systems execute tasks based on predetermined static or dynamic schedules. A static schedule does not change during the operation. On the other hand, a dynamic schedule relies on dynamic parameters that may vary based on scheduling decisions in each iteration. Dynamic scheduling is flexible and offers better resource utilization, but it is more complex with higher runtime.

Time-triggered system's behavior is deterministic. Predictable systems are a good fit for real-time and safety-critical systems. It makes the system applicable for industrial use-cases and the verification convenient. Time-triggered architecture is a suite of distributed algorithms for functions such as clock synchronization and group membership and their implementation in the form of TTA controllers, buses, and hubs. It is part of a comprehensive approach to safety-critical real-time system design that builds on time-triggered operation [1–4].

In the last few years, there has been an undertaking in the automotive industry to realize by-wire applications (brake-by-wire or steer-by-wire) without mechanical or hydraulic backup systems in vehicles. Therefore the electronic systems must be highly reliable and cost-effective to be feasible for mass production. Time-triggered architectures satisfy these requirements.

With the improvement of chip design manufacturing technology, the transistor density on dies increased drastically (e.g., making them smaller and stacking several

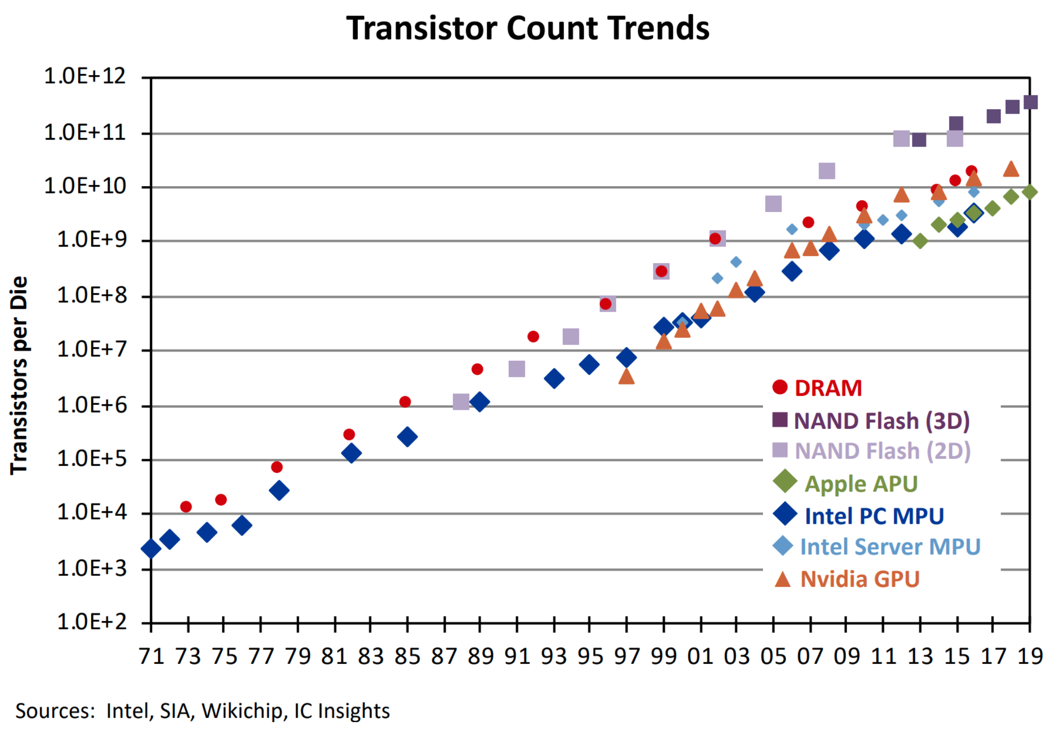


Figure 1.1.: Transistor count trends Continue to follow Moore's law (from [5]).

atop each other). The consequence is the more than ever packed transistors per chip area. Moore's Law predicted that the transistor count on a fixed chip area would double every two years due to shrinking transistor dimensions. Figure 1.1 shows the transistor count, which is still increasing exponentially.

To address the constantly growing demands on the available computing capacity and with advances in technology, platforms are gradually shifting from a distributed architecture with several different hardware components towards a centralized design with more minor but more powerful processing components. The fabrication of chips with multiple complex hardware components on a so-called Multi-Processor Systems-on-a-Chip (MPSoCs) brings advantages in terms of cost, space, and weight. Such central computing units typically consist of multiple microprocessors (e.g., ARM), reconfigurable logic (FPGA), and accelerators (GPUs) with complex interconnect hierarchies on one board, for instance, Xilinx UltraScale+ ZCU102. It also brings challenges to the integration of embedded software applications. The most crucial challenge is that another application must not influence the behavior of one application in terms of computation and communication resources. Currently, the standard solution is the time-triggered scheduling, in a simple case of time-division access, to periodically assign resources to tasks of the system for efficient resource management.

As chips become more populated with multiple heterogeneous processing elements, a scalable and flexible simulation infrastructure becomes more demanding. It prevents mistakes in the taped-out design and accelerates the time-to-market process. Designing and programming an MPSoC is a complex activity and debugging such a system is even more complicated because of its concurrent many-core nature. Therefore, a simulation and debugging infrastructure is necessary and makes this process convenient.

Providing a real-time response for the critical applications imposes the need to carefully simulate the system and its environment during the design time. A test environment to test the functionality and timing of complex multicore systems should enable the designer to test the software development and the functionality of the target platform before deployment in an acceptable amount of time.

1.1 Context

At the early design time, the correct functionality of the final system must be examined. This investigation includes testing the functional behavior implementation under the predefined time-triggered schedule together with an environment model. Functional testing becomes even more critical when the time-triggered architecture exhibits adaptive behavior to react to changing environmental conditions or scenarios [6]. Configuration and implementation of such a time-triggered system without any tool support is complex and could be another source of error. Implementing a time-triggered schedule that works correctly among different distributed components is very error-prone. For example, if a task performs memory access before the required valid data are received, this would lead to wrong computations. In addition, an incorrect configuration of messages' injection time could cause a significant delay in the safety-critical control cycle due to message collision on the communication medium.

Simulation of time-triggered systems differs from other kinds of system simulations. Since the notion of time plays an essential role in the functionality of the system, simply running the application on an Instruction-Set Simulator (ISS) is not enough. Time-triggered system execution follows a specific chain of activation and slack time that plays no role in a standard ISS. A more accurate simulation technique should guarantee that the simulation matches the real-world execution.

A fine-granular simulation at the clock pace can solve this problem, but it costs a considerable time overhead that a designer would like to avoid. Using the target implementation for testing (i.e., Hardware-in-the-Loop (HIL) simulation) provides the most accurate result. Although it gets accurate results and timing, it faces the weak observability concern that a host-based simulation was trying to solve.

In short, testing a time-dependent Design under Test (DUT) is a critical and challenging task, and the Electronic Design Automation (EDA) industry should provide more easy-to-use and reliable tools to facilitate the test and verification of such systems. This work facilitates the simulation-based verification approach for the time-triggered architecture to check if the system implementation fulfills its specifications, including real-time constraints.

1.2 Motivation & Problem Definition

Design and verification of time-triggered systems are becoming more difficult as the complexity of the systems grows, especially with the advent of multicore architecture and Network-on-Chip (NoC) communication. As these systems take control of safety-critical tasks, simulating and verifying them before deploying them on the hardware will lower the design's risk of having a bug or fatal problem. Relying on the mainstream approaches like Correct-by-Construction is not sufficient. There should also be a functional verification after the deployment of the system. It is because the formal analysis of the system is based on some ideal (primarily mathematical) semantics of a programming language. Nevertheless, the target platform may be compromised by various failures occurring at runtime (e.g., insufficient resources, soft error). A significant verification effort should be spent to guarantee the absence of such runtime errors.

Time-triggered architectures are commonly used in safety-critical systems due to their ease of verification and industrial applicability. With advancements in hardware architectures, the integration of such systems on a complex programmable MPSoC architecture is challenging, especially when there is a need to validate the interplay of functional behavior and timeliness properties. The available approaches to test these systems either use high-level functional models with annotated time or a hardware prototype in a HIL configuration. The former method does not represent the final implementation and undergoes reformulation of algorithms, data structure adaptation, and memory management alteration. While the latter approach suffers from costly changes and complex testing due to a lack of observability and debugging features.

Both Instruction Accurate (IA) and Cycle Accurate (CA) simulators can be used for functional verification. However, the preferred choice is a simulator that is as fast as Instruction Accurate and as time accurate as Cycle Accurate to correctly capture the component's interactions. The IA supports only a limited notion of time, namely the average execution frequency measured in Million Instructions Per Second (MIPS).

1.3 Scope & Research Questions

This thesis proposes a generic intermediate testing environment intending to achieve better observability and debugging capabilities to test time-triggered implementations and facilitate the development of complex time-triggered systems by automating the code generation. The proposed approach is called Globally Accurate Locally Inaccurate (GALI) a simulation model that combines Instruction Accurate simulation with predetermined time-triggered system configuration. It is an approach quicker than Cycle Accurate but more accurate than IA.

Our first and foremost goal in the project is the functional testing of the time-triggered scheduling for timing behavior. To do that, we test the fully binary compatible version of the partitioned system on an instruction accurate Virtual Platform (VP). The platform that we are focusing on in this work consists of tiles that are processor(s) executing time-triggered schedules that are connected via a time-triggered interconnect.

We claim a fast and accurate simulation through instruction accurate simulation by function timing back-annotation to avoid unwanted time drift. We formulate the following scientific questions regarding a functional test environment for time-triggered architectures in complex MPSoCs:

$RQ1$**:** How can the simulation of time-triggered systems on an instruction-accurate simulator be fast and sufficiently accurate to replace Hardware-in-the-Loop or Cycle Accurate simulation-based verification?

$RQ2$**:** What are the constraints and assumptions under which the proposed GALI technique works?

$RQ3$**:** Which properties shall be preserved in the GALI model abstraction concerning functional behavior, time-triggered schedule verification, and system profiling?

$RQ4$**:** Which verification step of a time-triggered system can be supported by the proposed GALI approach?

*RQ*5**:** Despite the timing inaccuracy (instruction-level) in the computation model, which profiling information can be obtained for system analysis?

*RQ*6**:** Which properties of the system have to be represented in the model so that the configuration flow can automatically generate the GALI model?

To summarize, this work addresses the central scientific question of how to provide a simulation model and the supportive design flow for the simulation of time-triggered systems. In particular, we use the natural features of time-triggered systems to tune up their simulation speed and reduce simulation time. Section 5 will revisit the scientific questions and formulate the contributions of the thesis.

1.4 Outline

The thesis is organized into three parts. The first part (Section 2 to Section 4) provides an overview of the foundations for this thesis, including background knowledge and prerequisites to understand better the flow and technique explained in part two. Section 5 presents our contribution based on the scientific context and the research questions mentioned before and compares our contribution to the related scientific work (Section 6).

The second part starts with Section 7 containing our GALI simulation technique. After specifying the modeling of the application, Section 8 describes the platform, mapping, and the implementation of them on the target platform. The tooling which supports the GALI simulation is also explained in Section 9.

The third part of this thesis contains the evaluation of the contributions and discusses the results. The evaluation of the integration flow is introduced in Section 10, and further explained in Section 11. Two different experiments, one hardware implementation in a Hardware-in-the-Loop (HIL) setup and another virtual platform in a Virtual Platform-in-the-Loop (VPIL) are compared against each other. We evaluate the accuracy of the GALI simulation technique by comparing the flight trajectory of a multirotor to demonstrate the difference between each flight behavior.

Finally, we assess the use of GALI model and conclude the thesis with a discussion of possible future activities in Section 12.

Part I

Foundations

System-Level Design 2

This chapter provides the foundations concerning the design, test, and simulation of time-triggered embedded systems. We first describe the V model used for the development process and how we have extended our tool flow based on the V model. It describes steps to be taken to design, implement, and test the system.

The virtual platform-based design methodology is explained next. This approach is used to test the generated binary code of the time-triggered system before deploying it on the hardware. Next, we take a closer look at the Synchronous Data Flow (SDF) Model of Computation (MoC) that helps us to execute the Time-Triggered (TT) schedule for the feasibility check-in first steps of our design flow. We will then present Correct-by-Construction design, which is the fundamental assumption in the GALI tool flow to ensure that the generated platform and the application are aligned with the TT specification.

At the end of this chapter, an overview of two different simulation technologies will be presented that can support the flow of design, verification, and validation.

2.1 Development Process

The verification and validation of requirements are a demanding part of a system design. The importance of testing has caused the traditional waterfall development cycle to be modified to create a new model called *V model*. In the traditional waterfall, the project cycle is usually displayed as a linear sequence of steps executed in a sequential manner separated by significant reviews. However, in the V model, every early development activity is linked to its corresponding testing activity. Several design phases begin in parallel, including requirements and proof of technical feasibility (prototyping).

As illustrated in Figure 2.1, the V model builds upon the traditional waterfall model by emphasizing the concept of *verification* and *validation*. Verification means static analysis (review) of the system development phase to check if the specific requirements are met. Validation means dynamic analysis (functional, non-functional)

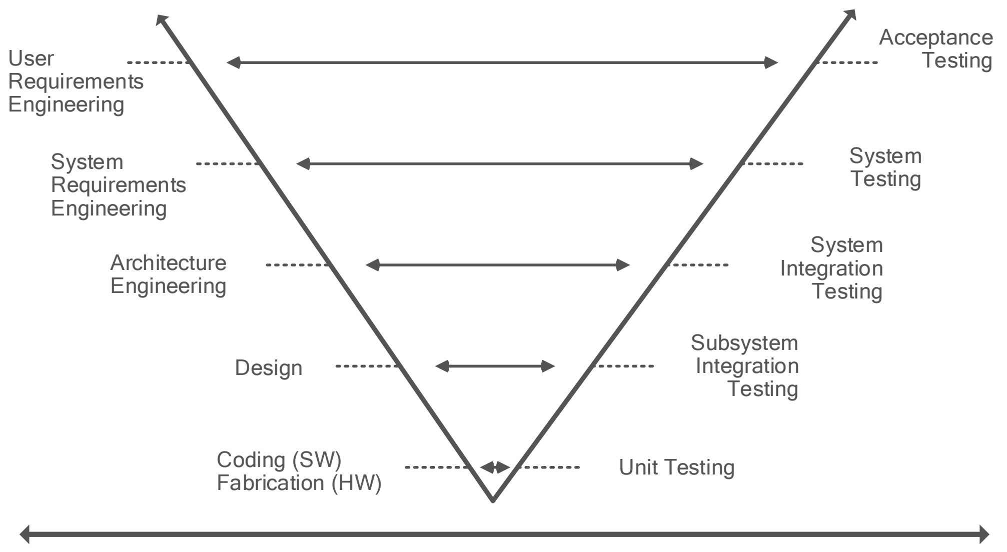

User Requirements Engineering
System Requirements Engineering
Architecture Engineering
Design
Coding (SW) Fabrication (HW)
Acceptance Testing
System Testing
System Integration Testing
Subsystem Integration Testing
Unit Testing

Figure 2.1.: V Model for Testing

of the system development by evaluating the system to determine whether the implementation meets the expectations and requirements.

To do verification and validation, the V model takes the bottom half of the waterfall model and bends it upward to make a V shape so that the activities on the right side verify or validate the result on the left side. More specifically, the development phase is planned parallel to the testing stage.

V model, like all other approaches, has both advantages and disadvantages. Nevertheless, the V model is still a convenient way of thinking about development and simplifying the abstraction. As it is shown in Figure 2.1, the model starts with the *Requirement Analysis* (also known as Requirement Gathering). It involves a detailed gathering of the requirements and expectations. Then *Architectural Design* should be performed. The requirements are broken down further into modules with different functionalities. In this stage, the data transfer and communication between the internal and external modules are being examined. Next is *system Design*, which involves designing the complete system, followed by software and hardware development for the product under design.

Afterward, on the right wing of the V model, *Unit Testing* is executed to eliminate bugs at the code or unit level. After completion of unit testing, the *Integration Testing* is performed. Submodules and modules are tested as a combined entity. Integration testing is associated with the Architecture Design stage. *System Testing* tests the complete application, including its functionality, dependency, and communication.

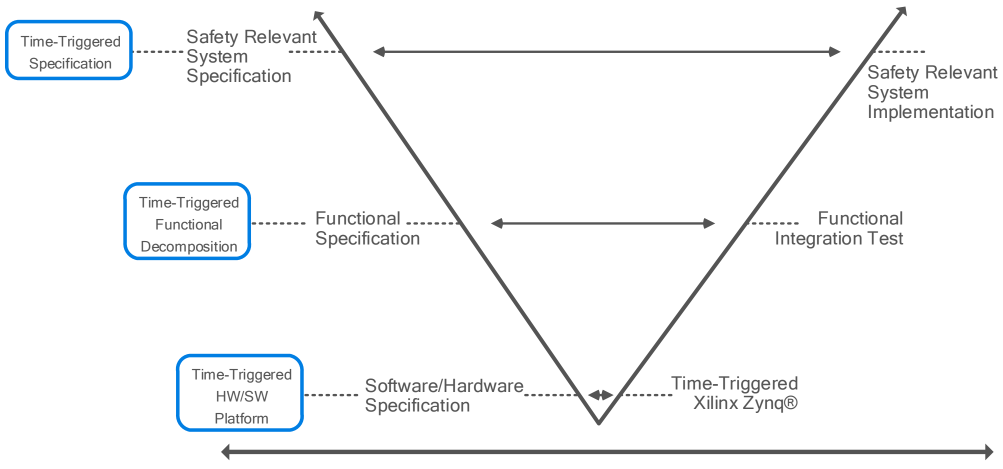

Time-Triggered Specification
Safety Relevant System Specification
Safety Relevant System Implementation
Time-Triggered Functional Decomposition
Functional Specification
Functional Integration Test
Time-Triggered HW/SW Platform
Software/Hardware Specification
Time-Triggered Xilinx Zynq®

Figure 2.2.: GALI V model

It also tests the functional and non-functional requirements of the developed application. Ultimately, the test is performed in a user environment that resembles the production environment. This *Acceptance Testing* is the final stage in the cycle and verifies that the delivered system meets all the user requirements and, if so, the design is ready to ship.

All the steps mentioned above have a specific output and a review process. Furthermore, the development phases on the left-hand side are performed parallel to their corresponding test phase on the right-hand side, and therefore, defects can be found at early stages.

We build the proposed development cycle in our design flow based on the V model (see Figure 2.2). All life cycle steps have to be supported with tools to handle the complexity of the systems, advance the development process, and improve reliability. The tooling topic is discussed in detail in Section 9.

- **Safety Relevant Specification $\leftrightarrow$ Safety Relevant Testing**

  As it is shown in Figure 2.2, it starts with the requirement specification of the time-triggered system. Specification of the requirement is the most error-prone phase if done manually by the user. The time-triggered specification (TT Spec) is a time-triggered schedule that includes start time, Worst-Case Execution Time (WCET), the system period, and the period of each task. The hyper-period of the system is constructed from the time-triggered specification. We check if the implemented design meets the specification of safety-relevant testing. Mistakes in this phase may lead to most faults and critical failures.

Most design errors are not low-level implementation errors but originate from the specification phase.

- **Functional Specification $\leftrightarrow$ Functional Integration Test**

  The next step is architecture engineering. In the functional decomposition phase, the design is mapped to the actual hardware, and each task and its timing behavior is assigned to the specific processing unit. In this phase, inter-processor dependencies that affect our design and communication infrastructure should be formed between processing units according to the time-triggered functional decomposition. For the testing phase of the validation progress, we can run an SDF representation (a data flow model represented as a graph rather than a sequence of statements) of our system to check if the scheduling and mapping concerning the dependencies meet all the requirements or not.

- **SW/HW Specification $\leftrightarrow$ Time-Triggered Xilinx Zynq**

  In this stage, the time-triggered implementation will be generated from the Software (SW) implementation mapped to the Hardware (HW) platform concerning the time-triggered specification. The result would be a periodic time-triggered system realized on the target board (a Zynq platform). Like all other stages, the implemented platform will be tested entirely through a Hardware-in-the-Loop setup that will be discussed thoroughly in Section 10.2.

2.2 Virtual Platform-Based Design

The process is delayed in the traditional board-based design methodology because of its sequential hardware and software development. The software developers can start the tasks in this methodology once the hardware becomes available. Considering several months of development for both HW and SW teams, the prototype is ready more than a year after specification.

Some bugs that were not seen during the hardware Testing will become visible during software development. There will be several iterations between the software and hardware teams to fix them. It can be possible that these bugs will be originated from different views and understanding of the HW and SW from the specification. It can further delay production.

Virtual platform-based design is one way to avoid these delays. In this methodology, first, a model of the hardware platform for software development is built [7]. This model is not used for the prototyping board. However, it gives the possibility to the SW team to start developing software for the target platform. This model is called *Virtual Platform* (VP).

Because abstractions used in developing a virtual platform take less effort and time to develop than a prototyping board. The internal micro-architecture implementation of components and processing elements is not necessary for the VP. What is needed, however, is that the VP must provide a programmable model of all software processors and functional models of all custom hardware components [7]. These models are usually at a level of abstraction higher than a cycle-accurate model but with visible registers and bus transactions. It helps debug and run-time analysis of the developed embedded software.

In VPs, processors are usually modeled in C/C++ and peripherals as remote function calls. Therefore, it provides the simulation speed essential for rapid embedded software development.

Another advantage of the VP is that it can serve as a common golden model for both HW and SW modules. And since the development can be performed in parallel, it shortens the overall prototyping time. However, a disadvantage of the VP is that any changes to the platform must be manually applied to the VP. A model-based design methodology like UML and SDF can solve this problem.

2.3 Model of Computation

A MoC is a method of representing system behavior in an abstract, conceptual form [7]. MoCs offer the advantage of being understandable by both humans and automated tools. They define the requirements and constraints of computations to be performed. MoCs are typically expressed in a formal manner using mathematical notation.

As it was mentioned in Section 2.2, Unified Modeling Language (UML) has become the standard modeling language for embedded system software. UML facilitates the modeling of system structure and provides constructs for representing control flow. However, data flow modeling is not well-supported in UML, especially for applications like Digital Signal Processing (DSP) and video systems, where data flow is a critical aspect of the system behavior. In practice, data flow modeling is often used in conjunction with UML, where data flow diagrams can be used to complement UML models to provide a more comprehensive view of the system

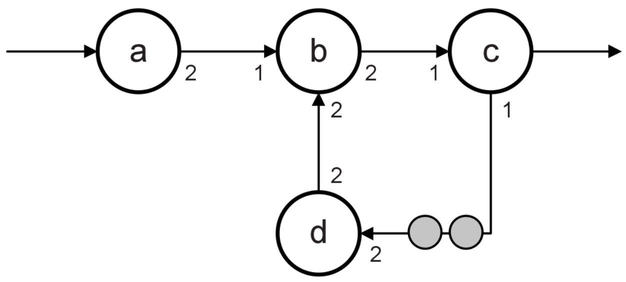


Figure 2.3.: Synchronous Data Flow (SDF) Example (from [7])

behavior. To model our system more comprehensively, we employ both state-based and data flow behavioral descriptions in this work.

SDF is an ideal modeling paradigm for the time-triggered domain. In this domain, all computation and data communication can be scheduled statically, ensuring that the implementation takes a finite time to complete all tasks and uses limited memory. This feature makes SDF suitable for applications that require periodic execution without the need for additional resources during runtime. This type of execution is particularly well-suited for applications such as digital signal processing, communication systems, and time-triggered systems. Additionally, SDF provides the advantage of determinism for time-triggered modeling. If the model is deterministic, the same inputs will consistently produce the same outputs.

SDF's determinism and static scheduling make it an attractive option for implementing safety-critical systems. In such systems, the correctness of the system's behavior is critical, and any deviations from expected behavior can lead to catastrophic consequences. By using SDF, the system designer can ensure that the system's behavior is fully deterministic, and the execution can be scheduled and analyzed statically. This level of predictability and determinism is crucial for safety-critical systems, and SDF provides an excellent framework for implementing such systems.

Overall, SDF's data-driven communication, static scheduling, and determinism make it a powerful modeling paradigm for implementing real-time embedded systems and safety-critical applications. SDF's suitability for time-triggered modeling makes it an ideal choice for applications that require periodic execution and deterministic behavior.

Figure 2.3 shows a simple SDF graph comprising four actors. In each iteration, actor `a` produces two tokens, while actor `b` consumes three tokens: one from actor `a` and two from actor `d`. Subsequently, actor `b` produces two tokens for actor `c`. Actor `c` consumes one of actor `b`'s tokens and sends one token to actor `d`. Finally, actor `d` consumes and produces two tokens on each of its input and output arcs. The two tokens on the arc between actors `c` and `d` are *initialization tokens* intended to resolve any deadlocks that may occur in the raw graph, which is the case in this example. Deadlocks can occur when there is a circular dependency between two or more actors, and each actor possesses an exclusive token that the next actor in the chain requires to proceed.

Timed Synchronous Data Flow is an extension of SDF that introduces a notion of time to the SDF model. Timed Synchronous Dataflow (TSDF) includes a timed data type with both a timestep and a carrier frequency attribute. In our tool flow, we utilize TSDF to simulate and test the time-triggered schedule for feasibility.

2.4 Correct-by-Construction Design

Correct-by-Construction is the enforcement of specification used in the early stage of the design through constraints. The philosophy behind it is to trade optimality for predictability. An advantage of Correct-by-Construction design is that it provides all the required functionality and guarantees the correct behavior, generating a predictable system. Furthermore, high-level optimization becomes more compelling in a predictable system due to the decreased error boundary. Correct-by-Construction is at the bottom a sequence of small, guaranteed-correct design transformations contrary to the more widely common *Construction-by-Correction* approach that encourages the integration of various phases of the design process into broad iterative loops [8]. Correct-by-Construction design techniques avoid the micro-engineering of every piece of the design that can become expensive in the design. Therefore, it is evident that Correct-by-Construction design techniques are worth applying. In Correct-by-Construction, the predictability comes with the necessary infrastructure and excessive resource consumption, which is a disadvantage. This means that high-performance design cannot benefit a lot from Correct-by-Construction.

To apply the Correct-by-Construction concept, the construction should begin with decomposing the system-level requirement into granular functions. Then, each function should be allocated to a component. This process builds a modular component architecture [9] that is easy to test and maintain. Correct-by-Construction does not mean that testing is not performed. However, testing is performed to validate the Correct-by-Construction process rather than finding bugs. Correct-by-Construction is at the heart of our approach, which guarantees that the binary generated by the tool exhibits the correct behavior. Any further changes to the specification will produce the same effect on the final product.

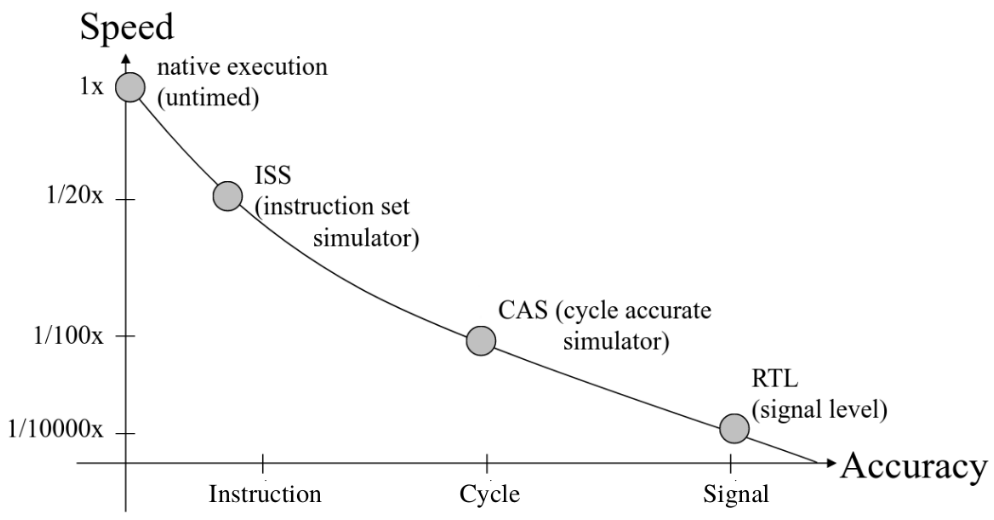


Figure 2.4.: Simulation Technologies (from [10])

2.5 System Simulation

Simulation is the most commonly used method to verify system models [7]. The design to be tested is described in a modeling language and is referred to as *Design under Test* (DUT). *Stimuli* which are a set of values that are input to the DUT, trigger a chain of events and computations in the DUT model. It is the job of the simulator to manage all these events and propagate them through the DUT. Propagation of the events through the DUT changes the values of various variables in the model. Whenever the variables' values are updated, a new event is generated to inform the simulator of this update. A critical input to the system is the clock signal. Based on the priorities of the simulation, different stimuli might have greater importance. Once the output of the DUT is updated at a given simulation time, the result must be checked if it is equivalent to the expected outcome.

Figure 2.4 shows different simulation technologies concerning their accuracy and speed in performing the simulation and obtained results. Based on the simulation's focus, one can choose the proper simulation technology. If the focus is on the accuracy, then the simulation should be as low-level as Register-Transfer Level (RTL). However, the simulation speed is decreased drastically. As the abstraction level goes upward toward the system level, the simulation speed increases exponentially. Two common major simulation techniques are instruction-accurate (IA or ISS) and cycle-accurate (CA or CAS). IA runs the program in batch mode without the knowledge of the clock cycles. CA, however, takes care of all the changes in the DUT state during the simulation.

2.5.1 Instruction Accurate Simulation

An instruction-accurate simulator is a simulation model which mimics the behavior of a processing system by reading instructions and maintaining internal states which represent the processor's registers [11].

This simulation technique is used in different situations, such as

1. Simulating the machine code of another hardware device or entire computer on a host machine.
2. Monitoring and executing the binary instructions on the same hardware for test and debugging purposes.
3. Improving the speed performance of simulations without using a cycle-accurate simulator.

The amount of detail and debugging capability that an Instruction Accurate simulator provides is lower than a Cycle Accurate simulator. And if the timing is essential, this simulation technique is not helpful due to a lack of clock cycle accuracy.

2.5.2 Cycle Accurate Simulation

Cycle-accurate simulators are a widely used method for computer-based simulations. They simulate systems in a clock-by-clock manner, accurately modeling the system's internal behavior. Specifically, a CA simulator models a system's micro-architecture by simulating it in discrete time-ticks that depend on the system's clock speed. This approach is commonly used in designing new microprocessors to accurately simulate, test, and debug the entire system, including its operating system, compilers, and utilities, before physically manufacturing the chip.

However, cycle-accurate simulators are typically slow, sometimes orders of magnitude slower than other simulators, but they provide accuracy at the clock cycle level. To mitigate the slow speed of this simulation method, some researchers use techniques like "reduced execution" [12] or simulating only a specific part of the system. However, using these speed-up techniques may result in errors and poor accuracy.

Despite their usefulness, cycle-accurate simulators have several disadvantages and limitations that restrict their widespread use in various applications. One major drawback is that they are usually slower than other simulators, which can limit their practical utility, especially when simulating large and complex systems. Another limitation is that cycle-accurate simulators are often closed-source, making it challenging to validate their accuracy and correctness against real hardware. Finally, the more general and configurable the simulators are, the less accurate and reliable their results tend to be. These limitations make cycle-accurate simulators less than ideal for many real-world applications.

In summary, while cycle-accurate simulators are useful for accurately modeling the behavior of a system at the clock cycle level, they are often slower than other simulators which limits their practical utility, particularly when simulating large and complex systems. Additionally, their limitations, including being closed-source and less accurate and reliable as they become more configurable, can restrict their widespread use in various applications.

3 System Architecture

In this chapter, the time-triggered architecture is described. However, our system architecture (described in Section 7) is a subset of time-triggered architecture, and for simplicity, many detailed communication protocols (e.g., fault tolerance) are not considered. We only consider the time-triggered system, which means every action in the system is performed based on a predefined static schedule.

3.1 Time-Triggered Architecture

Safety-critical systems that are based on time-triggered architecture (TTA) are more convenient to design and verify than other available architectures [14]. The critical characteristic of time-triggered systems is that all significant events must adhere to a predetermined schedule, including tasks and messages. The time-triggered paradigm of a real-time system is based on the unique view of the world: time-triggered systems are not driven by the events that happen in their environment, but they decide when to look at different events based on a global schedule [15].

Time-triggered architecture is a set of state machines triggered by the progress of the time. A time slot and specific resources are available in each state, and it is guaranteed that while the system is in one state, no collision between different states can happen. In time-triggered architecture, the time is partitioned between different tasks, and the designer dedicates a timing slot to each task. A task must start at a specific time and should not take more than a defined length (i.e., WCET). Each task in the task set can either be a computation (Section 3.2) or a communication (Section 3.3) task which is repeated identically in each cycle.

This model of computation (MoC) and communication guarantees the timeliness of the system. This feature makes time-triggered architecture uniquely suited for the complex safety-critical system by bringing the separation between different computation and communication tasks. Time-triggered architecture consists of several *nodes* connected. A node is a combination of a host and its time-triggered architecture controller. This modularity implements the partitioning feature and guarantees that a fault in one node of time-triggered architecture, or one application

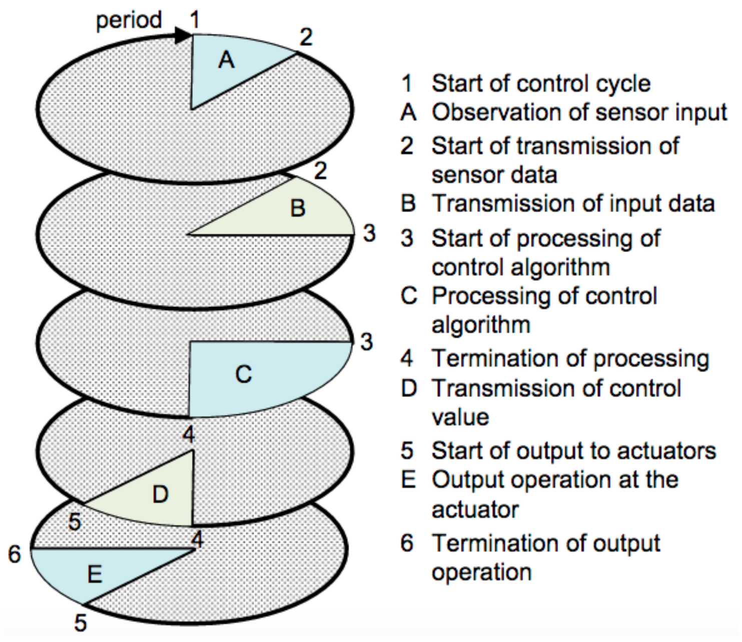

period
1
2
A
2
B
3
3
C
4
D
5
4
6
E
5
1 Start of control cycle
A Observation of sensor input
2 Start of transmission of sensor data
B Transmission of input data
3 Start of processing of control algorithm
C Processing of control algorithm
4 Termination of processing
D Transmission of control value
5 Start of output to actuators
E Output operation at the actuator
6 Termination of output operation

Figure 3.1.: Example of cyclic execution of a control loop in time-triggered architecture (from [13])

supported by time-triggered architecture, does not propagate to other nodes and applications.

The example taken from [13] gives more insights into the time-triggered architecture. Figure 3.1 demonstrates a control loop realized by three components of A) sensor data acquisition, C) control algorithm processing, and E) actuation. It shows the periodic execution and temporal alignment of different tasks in the time-triggered architecture. The perimeter in this cyclic model of time representation represents the duration of tasks' periods.

Although the time-triggered strategy provides many benefits, it may also lead to resource waste when fully implementing the time-triggered architecture. Also, while time-triggered schedules excel at determinism and jitter control, they are hard to design and lack flexibility.

3.2 Time-Triggered Computation

The time-triggered architecture provides a computing infrastructure for safety-critical real-time control systems. It focuses on time-triggered operations and provides services to guarantee applications' timeliness. The time-triggered model of computation

(MoC) is based on dividing an extensive distributed system into almost autonomous subsystems with an interface between these subsystems [16].

The four building blocks of a time-triggered MoC is as follows [16]:

1. An *interface* acting as a boundary between subsystems. It consists of a memory element that is shared between two subsystems. An interface can be viewed as a dual-ported memory.
2. A *communication system* which is a medium that connects interfaces. It can be imagined as a train between stations with a timetable about the departure time and expected arrival time.
3. A *processing element* that reads and writes to the interface. It encapsulates one or more processors, including a memory, system, and application software.
4. A *transducer* connects a real-time entity in the environment to an interface and vice versa. It models the input/output system of a real-time system.

These elements have access to a globally synchronized time base with sufficient precision.

3.3 Time-Triggered Communication

A time-triggered system not only schedules activity within nodes it also handles the reliable transmission of messages between nodes. The predefined static schedule dictates the starting points of all communication tasks.

Time-triggered architectures rely on a communication network to realize a reliable distribution. None of the commonly used communication systems in vehicles meet the safety-related systems requirements. They all do not guarantee determinism. The Time-Triggered Protocol (TTP) is designed for safety-related applications and fulfills the following criteria [17]:

Membership service; every node knows about the actual state of other distributed nodes.

- Fault-tolerant clock synchronization service (global time-base).
- Mode change support.
- Distributed redundancy management.

Time-triggered architecture does not use dedicated wires to communicate clock reading. It exploits the fact that communication is time-triggered according to a global schedule. When one node receives a message from another node, it records the local clock. It subtracts a fixed network delay, which indicates the difference between this adjusted clock and the global schedule, indicating the skew between both nodes' clocks. The time-triggered nature of the protocol means that each node expects a message from other nodes at a specific time and, therefore, can detect a dropped message.

Time-triggered architectures can have a different implementation. In the Time-Division Multiple Access (TDMA) method, it operates as a broadcast bus. The global schedule allocates a slot for nodes on each repetition of execution. In the Network-on-Chip (NoC) method, each node is connected via an interface to a switch of the network. Each interface has a schedule that indicates the time to send a message and expect a message in its inbox.

4 System Verification and Validation

As mentioned in Section 2.1, the V model is used to design, develop and test high-quality systems. The V model emphasizes the concept of verification and validation. The ultimate goal is to generate an accurate and credible system.

Verification checks if a system meets a set of design specifications. It involves performing tests to model or simulate a system and then analyzing the results. Verification is performed during the development and also the post-development phase.

Validation ensures that a system meets the user's operational needs. It involves using simulations to find faults or gaps that might cause invalid or incomplete verification or development of a system.

Informally speaking, validation can be expressed by the question "Are you building the right thing?" and verification by "Are you building it right?" It is possible that a system passes the verification phase but fails under validation. A sample scenario can be when a system is built as per the specifications, but the specifications themselves fail to address the user's needs.

Here we introduce different V&V (Verification and Validation) techniques. There is no one solution to check everything. Different approaches are needed, as they complement each other in their purpose and capabilities at different levels of abstraction and various phases of the development.

4.1 Formal Method

Formal methods assure the safety-critical claims to the developers and users of the time-triggered architecture. The proofs depend massively on *arithmetic reasoning* [18]. The correct operation of components is generally formalized in terms of assume-guarantee reasoning.

Formal verification is intended to prove the absence of errors. To perform a formal verification, it is required to have adequate modeling at an acceptable level of abstraction. Therefore, the impact is limited by the model accuracy, which typically neglects most of the hardware-specific effects and anomalies [19].

4.2 Simulation-Based Approach

It is the most commonly used verification approach. During simulation-based verification, the DUT receives the input stimuli from the testbench. Then the output from the DUT is compared with the reference (golden) output.

Before simulation a design, it goes through a static code analysis tool to flag errors, bugs, stylistic errors, and suspicious constructs. It checks static errors, errors that can be uncovered without input stimuli. Then, stimuli vectors of the inputs are generated. These inputs are both designed inputs and pseudo-random tests to explore all the aware and unaware areas of the design. When the test vector is ready, the simulator performs the simulation. A simulator can be an event-driven, cycle-accurate, or hardware simulator. The quality of simulating a test on a design is measured by the *code coverage* and *fault coverage* the test provides. The coverage measures how much the design is stimulated and verified.

To be able to verify and test the whole embedded system, a simulation technique should be used to test hardware (HW) and software (SW) components together. This approach is called co-simulation, where the integration and the synchronization of HW and SW modules require a permanent control of consistency and correctness. Co-simulation is ideal for checking the presence of expected behavior. However, it is impossible to have an exhaustive co-simulation. It is not adequate to check the absence of erroneous behaviors, which is the goal of verification methods [19].

4.3 Hardware-in-the-Loop

Hardware-in-the-Loop (HIL) bridges the gap between testing a model and the actual implementation by incorporating the dynamic behavior of the physical prototype. HIL provides the possibility of replacing incomplete and/or inaccurate models with the real-world counterpart. It is an actual hardware setup coupled with simulation software for testing the hardware and software components under realistic conditions.

A HIL setup usually has three main components:

1. The environment simulator

   It models the environment and provides the needed I/O ports to reproduce the behavior of the simulated system under desired conditions—a communication medium exchanges data between the simulator and the hardware under test.

2. The hardware design under test (DUT)

   It helps test the actual physical controller device connected to the controlled environment.

3. The software partition

   The SW partition of the system is tested running on the actual hardware together in the simulated environment.

HIL has, however, drawbacks and limitations, especially missing the flexibility in designing and performing test scenarios like in the simulation. Also, when integrating HIL to the co-simulation, it is critical to make sure that the data flow is synchronous between different components of the setup. The synchronization strongly influences the accuracy of the experimental result. Nevertheless, ensuring synchronization is not always an easy task since HW runs much faster than the environment simulator. These two should be synchronized. Also, the connection medium delay should be calculated and considered.

Despite HIL being very accurate, it lacks enough visibility because of the limited capacity of capturing internal signals of the hardware. Also, the complexity of making the setup is non-linear when adding new features.

5 Thesis Contribution

In the previous chapters, we have provided an introduction to the domain of time-triggered architecture and covered background knowledge, basic methods, and current challenges. In this chapter, we present the contributions of the thesis concerning the scientific questions discussed in Section 1.3.

5.1 Contributions

The following contributions aim at improving the flow of design, test, and simulation of time-triggered systems.

Contribution $C1$: We propose a novel simulation technique called Globally Accurate Locally Inaccurate (GALI), to speed up simulation of time-triggered architecture using instruction accurate simulators (see Section 8.2).

The proposed simulation technique starts with the specification of the time-triggered architecture and its features that help us speed up the simulation by using instruction accurate simulation technique. Contribution C1 is the approach to joining these two (TTA and IA simulation) to perform a fast and accurate simulation in the domain of time-triggered architecture. This contribution answers the $RQ1$ scientific question.

Contribution $C2$: We provide a systematic way to define and configure system architecture, tasks, and communication channels in our proposed simulation technique. It is called GALI Configuration Environment (GALI-CE) (see Section 8.3).

To answer the question about the design flow and the properties that help in model-based simulation, we have developed a front-end to facilitate task definition and specification, correct-by-construction system implementation, and hardware project generation. This contribution is linked to the $RQ2$ scientific question. GALI-CE considers, under the hood, all the constraints and assumptions under which the proposed GALI technique works.

Contribution $C3$: We propose evaluation of different backend implementations (see Section 8.3.3). GALI Configuration Environment (GALI-CE) automates cross-platform transformation for native host compilation, ARM, and Microblaze processing elements.

GALI configuration environment front-end offers generic system definition independent from the target processor. The designer can test and verify the design with the native flavor, also known as host-based simulation. After successfully passing the native simulation, then the designer has the option to provide the cross-platform toolchain, and the tool can handle the configuration and compilation. This contribution answers the $RQ4$ and $RQ6$ scientific questions. GALI-CE provides the infrastructure for future work to perform power and performance profiling which is explained more in Section 12 and answers the $RQ5$ research question.

Contribution $C4$: The evaluation setup (described in Section 10.2) is also a contribution to this thesis. This setup has helped the SAFEPOWER project to evaluate the proposed concepts and then further improve to address the issues in this thesis.

It has also won the first prize in the Xilinx Open Hardware 2019 competition in the Ph.D. category. Many hours of this thesis have been dedicated to preparing the setup. It consists of simulator coupling, configuring the flight simulator to be able to talk to the board, developing an adapter board to imitate the sensor and actuator interface, and a real-time co-simulation synchronization was also introduced into the setup. This contribution evaluated $RQ1$ to $RQ6$ possible.

5.2 Assumptions & Constraints

The contributions mentioned in the previous section contain the following assumptions and constraints regarding the hardware, application model, platform architecture, and schedulability. We have evaluated them by answering the $RQ3$ scientific question.

$A\&C1$: The platform consists of multiple tiles (a.k.a. nodes). Each tile can contain multiple Processing Elements (PEs) and shared and private memory.

$A\&C2$: Processing elements (PEs) are timing predictable. Time predictability is necessary to verify the correct operation of a time-triggered system. It refers to the ability to precisely calculate the duration of actions on the system (the predictability of the Worst-Case Execution Time and the Best-Case Execution Time).

$A\&C3$**:** The communication infrastructure is also timing predictable (for instance a TDMA interconnect such as a predictable NoC).

$A\&C4$**:** The application model consists of a set of tasks and communication channels. Tasks communicate via a uni-directional virtual channel that transports a message between sender and receiver tasks.

$A\&C5$**:** Tasks and Messages are characterized by the pair of execution time/injection time and worst-case execution and transport delay belonging to the processor it is assigned to, and therefore no interference.

$A\&C6$**:** The schedule is static, predetermined, and timing deterministic and is given as an *input* to our design flow. The schedule consists of repetitive tasks, which will form a hyper-period to be executed each time.

$A\&C7$**:** Instruction-Set Simulator for the target platform should be available.

Detailed assumptions and constraints on application, platform, and mapping model, which are acceptable by GALI are explained in Section 7.

Related Work 6

Available approaches to test time-triggered systems use primarily high-level functional models with annotated time (to express the time-triggered (TT) schedule) or Hardware-in-the-Loop (HIL) prototypes with precisely reproducible timed environmental simulation models. High-level functional models do not represent the final implementation and may undergo reformulation of algorithms, an adaptation of data types, or changes in memory management. Thus, these models do not represent the functional implementation deployed on the actual hardware platform. On the contrary, the low-level HIL models are based on the final product, where changes are very costly, and testing and debugging are complicated on the target hardware platform. Time-triggered systems can be simulated with cycle-accurate processors and full system simulators, but it takes enormous time. Hardware models may be specified as low as at the RTL level simulation (Hardware Description Language (HDL) level with nanosecond resolution), making the simulation usually take days to run a large simulation. In the following, we divide our contributions into two categories and compare them with different research results and try to highlight our contributions.

6.1 Generating Configuration

Plenty of works address modeling frameworks to help the system designer configure the system. Xoncrete[20] for instance, is a scheduling tool that provides a user-friendly interface for capturing and editing all elements of a partitioned system. Its primary purpose is to generate configuration files compatible with the XtratuM hypervisor. Work presented in [21] maps synchronous data flow model to time-triggered architecture to develop an optimized scheduling configuration. Both works follow the ARINC 653 time and space partitioning mechanism.

There are other frameworks to study the temporal behavior of applications to see if they meet the system constraints. For example, Cheddar[22] is a framework for designing and testing custom schedulers. MAST[23] is an open-source tool for modeling real-time applications and performing timing analyses of those applications.

RapidRMA[24] is an analysis and simulation scheduling tool for different scheduling methodologies.

Another direction of work generates code from a high-level model. For example, [25] is an off-line mapping tool that allows scheduling and automatic code generation for time-triggered platforms. [26, 27] are UML/MARTE based model which enables capturing the set of possible design solutions abstractly and graphically by relying on UML and the standard MARTE profile. The framework produces an executable functional code of the application components.

UML modeling environment and automatically generated SystemC code from UML designs are explained in [28]. Also, [29] uses SystemC-based modeling and simulation framework for time-triggered safety-critical embedded system.

The framework in [30] provides formal specification and formal verification of a time-triggered protocol without any functional code. The work in [31] uses high-level modeling for verification of the system design via functional simulation, fault modeling, functional test, and fault injection.

Our framework does not perform scheduling or scheduling optimization. We assume that the schedule is given and all the values are correct. It can, however, check if the system's functionality concerning that timing is still consistent or not. Also, we derive no code from a high-level model such as UML, the model that we have is our canonical architecture that the user code would instantiate. Our original simulation approach at the core of the modeling framework distinguishes our work.

6.2 Simulation of time-triggered Systems

There are two approaches to speed-up system simulation:

1. *Dynamic Translation* of target processor instructions to simulation host instructions.
2. *Native Simulation*, where the target code is statically analyzed and transformed into a host-compilable code where timing properties of the target platform can be back annotated.

Native simulation can be faster (factor 10-100) than dynamic translation. In contrast, on the downside, native simulation approaches have difficulties supporting target platform traceable debugging because the original binary code is not retained. Our approach requires annotating the functions with the time-triggered scheduling information at the instruction level, so we cannot use a native simulation approach. Instead, we use existing time-triggered functional simulation approaches and more generic approaches that include timing information into functional and instruction-accurate simulation models.

The work done in [32] is very similar to this work. However, the main difference is the simulation platform. The actual processor model (i.e., ARM or Microblaze) and final binary files are used in our work. However, [32] executes the tasks on the simulation Personal Computer (PC) and assumes that there is a strong correlation between execution time on PC and the actual embedded processor. In the GALI model, the simulator works at the instruction level. Therefore helpful information regarding functional and extra-functional properties can be extracted and used further. Another difference is the underlying architecture. GALI targets time-triggered architecture, which works based on target ticks and a predetermined schedule. It does not divide the system Into individual cyber and physical parts and, unlike [32] does not only keep the data and time correctness only at the physical interaction points, but it is synchronized after the end of tasks because of the time-triggered nature. Synchronization points should not always be software and hardware interaction points. GALI simulates the complete system and is not just focused on the cyber systems.

An exciting work that aims at abstracting timing information to a minimum required amount is called Result Oriented Modeling (ROM). The approach has been successfully applied to the functional simulation of on-chip communication [33]. The main goal of ROM is to produce a correctly timed result of a process as fast as possible. ROM uses the fact that internal states in the communication are not interesting for the caller. Therefore, it omits the internal forms entirely and optimistically predicts the result. It helps to optimize the unnecessary details by ignoring intermediate states. It reduces the amount of computation and, thus, increases the execution performance. Our work is, to some extent, similar, as we do not rely on a detailed processor timing model to be executed during target processor instruction simulation but apply timing information from the time-triggered configuration in the functional simulation. We need to use a mapping between Observable system states and pre-known time-stamps in line with ROM. The essential part of our goal is to produce only the result of the process, not intermediate timing states. Like SystemC TLM *wait-for-time* statement, our work assigns the execution time, the actual time needed to execute a function, set of Functions, or tasks.

A SystemC simulation framework for time-triggered architecture that provides a modeling framework for safety-critical systems is introduced by [34]. It presents a

SystemC-based extension for modeling generic time-triggered architecture-based safety-critical embedded systems called Executable Time-Triggered Model (E-TTM). There are a couple of incremental works using SystemC-based modeling and simulation like [34, 35], which are all SystemC-based functional extensions for the modeling of time-triggered architecture. [36, 37] are also focused on TLM-based approaches. The main shortcoming of these approaches is that there is no final target binary available, and it completely abstracts from a specific target execution environment. There are also papers on co-simulation frameworks for simulation of systems, which use a combination of different simulation technology for the other parts of the system. For instance, [38] uses SystemC to model the cyber part of the platform, which forms the backbone of the virtual prototyping, and TTEthernet [39] enables TT communication systems. The co-simulation approach resembles the final product better than pure SystemC simulation but still has the same disadvantages.

For applying simulation technology on time-triggered systems, accurate modeling of the system's temporal behavior should be supported. The correct functional behavior of time-triggered systems is inextricably connected with a functionally aligned progress in time. Instruction-accurate (IA) simulators are fast, but they insufficiently capture timing events because of abstraction. Therefore, they are not suitable for testing time-triggered systems. For instance, Simics[1], QEMU[2], and OVP[3] are instruction-accurate simulators and have A limited notion of time, and therefore not suitable for a time-triggered system simulation. Gem5[4] can simulate both instruction-accurate (IA) and cycle-accurate (CA) models. Gem5, in cycle-accurate mode, can simulate a time-triggered system accurately. However, it is too slow [44]. The same applies to ZSim[5]. OVP QCA (Quasi-Cycle Accurate) [46] is aimed at improving timing accuracy. A processor timing model improves the OVP simulator's timing accuracy. This technique can be applied on time-triggered system simulation, but the simulation time overhead is significant due to the complexity of the Quasi Cycle Accurate (QCA) timing model.

The GALI approach tackles the shortcoming of instruction-accurate simulators by producing the accurate end-result of the IA simulators concerning the start and end

[1]Simics[40] is a processor and complete system simulator that is capable of executing the target processor binary, which focuses on functional software validation.

[2]QEMU[41] is a machine emulator that uses JIT to execute unmodified target processor code on the host computer.

[3]OVP[42] uses similar JIT technology as QEMU but has been designed initially for fast embedded microprocessor simulation.

[4]The gem5[43] simulator provides four different CPU models, each of which with different speed versus accuracy trade-offs.

[5]ZSim[45] is a micro-architectural simulator that uses dynamic binary translation and has s simple cycle-per-instruction timing model and a cycle-approximate timing model that uses program timing instrumentation.

of the execution time of an event. However, it ignores the unimportant detail of timing between start and endpoint (similar to the ROM[33] approach mentioned above). To the best of our knowledge, GALI allows for the first time to combine instruction-accurate simulators with time-triggered system functionally, as there is no support for time-triggered static scheduling architectures in available instruction-level simulators. GALI can be applied to all the well-known simulators.

Part II

Design Flow & Modeling Approach

System Model

7

The previous chapters have emphasized the importance of time-triggered systems in safety-critical systems and the need to have a platform for verification and validation of the system in the early design stage. This chapter describes the system model definition and the proposed design flow for time-triggered embedded systems on MPSoCs. Its goal is to provide a test and evaluation environment during the design phase such that the designer can verify and validate the given static schedule, mapping, and implementation decisions.

Based on the GALI V model described in Section 2.1 (Figure 2.2), the first step in the development process is to capture the requirements of the application. The design process typically starts with a specification model to express the requirements of the embedded system in terms of functional behavior and physical constraints.

We opted for a hardware platform with timing predictable processing elements and a timing predictable communication infrastructure (for instance, – but not limited to – a predictable NoC). A platform consists of multiple tiles (a.k.a. nodes). Each tile can contain multiple Processing Elements (PEs), shared and private memory. We assume that every task and message is characterized by its WCET for the processor it is running on, as we do not consider processor interference [47]. Each communication between tasks on different processors is realized by a Virtual Channel (VC). A VC is an abstraction of a directed connection between two ports of two communication interfaces. It is mapped to a time-triggered predictable communication fabric, and a time-triggered communication interface [48]. The communication timing's properties are characterized by a message injection time and a worst-case transport delay (i.e., Worst-Case Response Time (WCRT)). Figure 7.1 depicts an overview of our system model. To better understand such a system, first, we describe the three elements of the model:

- **Application** models the application consisting of tasks and their connections (Figure 7.1-a).
- **Platform** models the TTA hardware platform (Figure 7.1-b).
- **Mapping** represents the synthesis decisions of the software to the hardware (Figure 7.1-c).

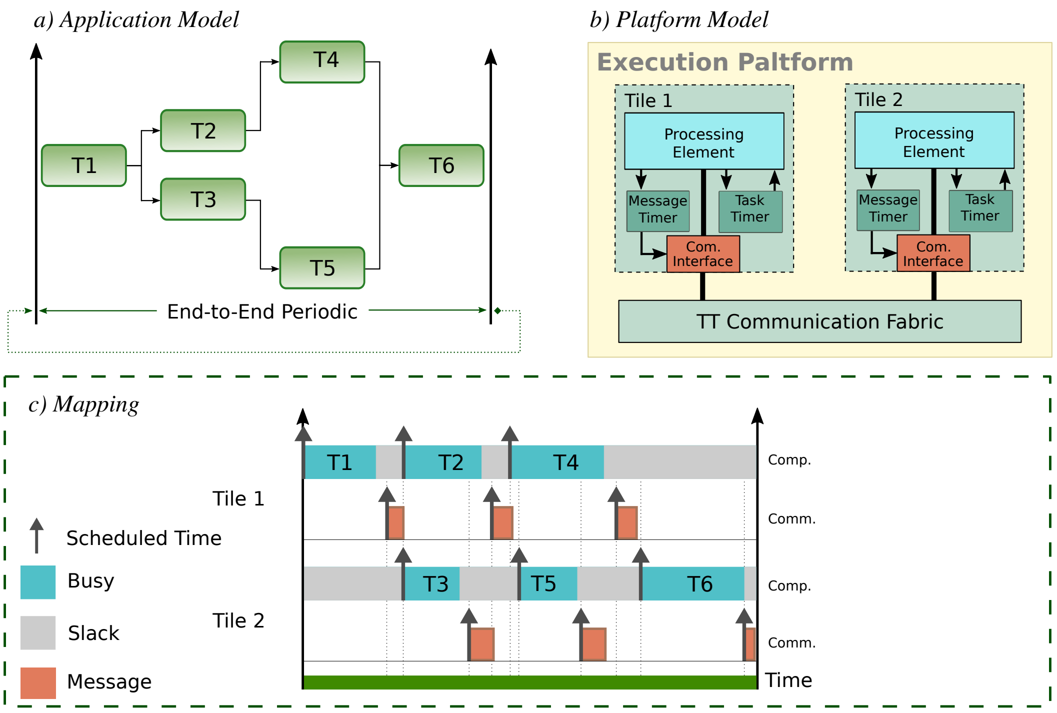


Figure 7.1.: Overview of the System Model

7.1 Modeling the Application

As shown in Figure 7.1-a, the application model consists of a set of `Tasks` (T_1 to T_6) and communication channels (arrows). The communication `Channels` between tasks represents a uni-directional channel that transports a message between a sender and receiver tasks.

In formal language, the *application model* A_i is defined as a tuple $(\mathcal{K}, \mathcal{C})$ consisting of:

1. A set of tasks $\mathcal{K}$, where a tuple characterizes each task (p, ϕ, τ), where p is the period, ϕ is the offset time of the task (i.e., the time elapsed between time 0 and the release of the first instance of the task) and τ is the task Worst-Case Execution Time. An obvious assumption is that: $\forall k \in \mathcal{K} : \tau_k \leq p_k$.
2. A set of $\mathcal{C}$ communication channels between tasks where $c_j \in \mathcal{C}$ is defined as $c_j = (k_{o_j}, m, k_{i_j})$ representing an uni-directional channel which transports a message of unique type ($m \in \mathcal{M}$) between a sender task k_{o_j} and a receiver task k_{i_j} (where $k_{o_j}, k_{i_j} \in \mathcal{K}$ and $k_{o_j} \neq k_{i_j}$).
3. A relation $\mathcal{E} : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{K}$ denoting the directed communication edges transporting messages between tasks where $\mathcal{E}(k, m) = \{k' \in \mathcal{K} | \langle k, m, k' \rangle \in \mathcal{E}\}$

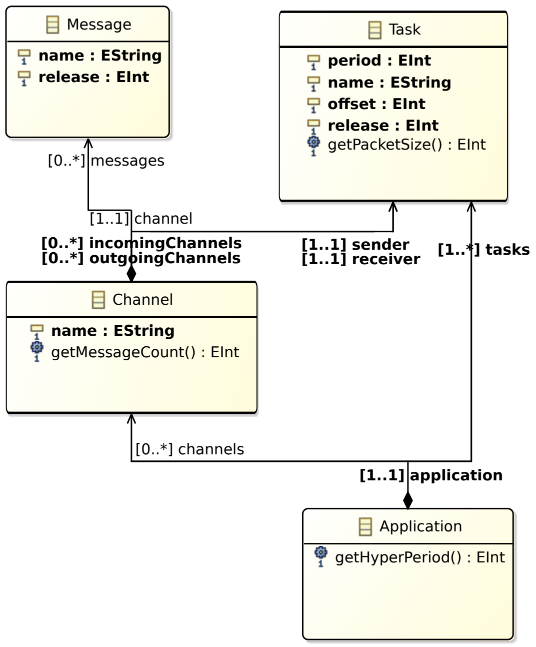


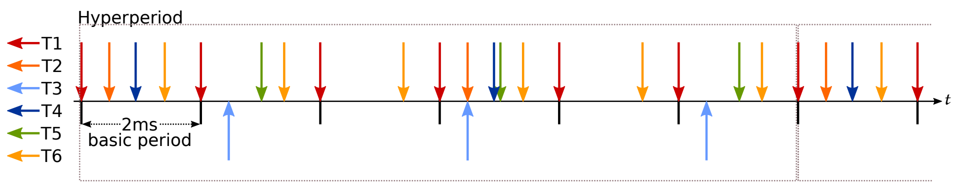
Figure 7.2.: UML Diagram of Application

indicates the number of messages ($m \subseteq \mathcal{M}$) transferred between sender task k and receiver task k'. $\mathcal{E}$ is realized throughout our work as virtual channels (VC).

Both tasks and messages have a specific timing associated with them which determines when these tasks have to be executed and when the messages have to be sent. Figure 7.2 illustrates the connection between `Message`, `Task` and `Channel`, and how they are related to the `Application`. Three parameters describe the timing features shown in the UML diagram for the task:

- **Period** determines the time between the starting points of two invocations of a task.
- **Release** time describes when the task starts in its period.
- **Offset** determines how many *basic periods* or iterations the task should skip until its first release.

The *basic period* is defined as the minor period in the set of all tasks, representing the frequency at which the system meets its control timing requirement. The *hyper period* H is the time after which the pattern of the periodic tasks repeats itself (i.e., a cyclic schedule representing the maximum time interval between two consecutive completions of execution). It specifies the time for all tasks to get executed at least



	T_1	T_2	T_3	T_4	T_5	T_6
Period	2	6	4	6	4	2
Offset	0	0	2	0	2	0
Release	0	0.37	0.37	1	1.05	1.5

Figure 7.3.: A Hyperperiod of given tasks T1 to T6 – Values are in ms

once. The hyper period is defined as the least common multiple of all tasks' (of a total number of n) periods:

$$H = lcm_{i=1}^{n}(p_i)$$

Figure 7.3 shows tasks T_1 to T_6 of our illustrative example that we use throughout this paper. It assumes that the system is working at a frequency of $500Hz$. Therefore it has a $2ms$ basic period. Period, Offset, and Release time of tasks indicate how they appear in a hyperperiod. The period parameter shows how often a task repeats. It is multiple of the basic period. The hyper period of all tasks is therefore $lcm(2, 6, 4, 6, 4, 2) = 12ms$. The offset shows how much time a task is shifted in one hyperperiod for the first execution. The reason for having an offset is that there are tasks that should wait for the input of other previous tasks. Therefore instead of starting right from the first period, they need to wait for one or more periods. The release time shows when, within the basic period, a task should be executed/released.

7.2 Modeling the Platform

As mentioned earlier, Figure 7.1-b shows one example of the supported architecture. It consists of two processing tiles connected via a communications subsystem. Each Processing Element (PE) has a timer to set the deadline for the next task. Each communication interface also has a dedicated timer controlled by the PE to enable the time-triggered communication functionality.

Figure 7.4 depicts the platform model. The main element of the execution platform is defined as $EP = (\mathcal{T}, \mathcal{CS}, \mathcal{VC})$ consisting of:

1. A finite set $\mathcal{T}$ of tiles.

2. A shared time-triggered communication subsystem $\mathcal{CS}$.

3. A finite set of virtual channels ($\mathcal{VC}$) which can be mapped to one physical communication channel (e.g., in the case of a bus) or to multiple dedicated physical channels of a $\mathcal{CS}$ (e.g., in the case of a Network-on-Chip) transporting messages between tiles.

A tile is defined as $T = (PE, I)$ consisting of:

1. A processing element $PE = (PE_{type}, f, \delta_{PE})$ with local data and instruction memory, where PE_{type} is the type of the processor and f is its clock frequency. δ_{PE} is the PE schedule, configured at design time, that assigns each task $k \in \mathcal{K}$, which is mapped to this PE, on a computation time slot $\delta_{PE} : \mathcal{K} \rightarrow SL_{comp}$.

2. A communication interface $I = (\mathcal{O}, \mathcal{A}, \delta_I)$ that is connected to the PE through a local bus and connected to the communication resource on the other side. $\mathcal{O}$ is a set of output ports, and $\mathcal{A}$ is a set of input ports. Each port is realized as a buffer that can store one message and is accessible through memory-mapped I/O. Output ports store a message that will be sent through the communication subsystem to an input port in another tile's communication interface. δ_I is the I's schedule, configured at design time, that assigns each virtual channel $vc_a \in \mathcal{VC}$ that has its origin in this communication interface (i.e. $vc_a.p_o \in \mathcal{O}$) to a communication time slot $\delta_I : \mathcal{VC} \rightarrow SL_{com}$.

A Virtual Channel $\mathcal{VC} = \{vc_1, vc_2, \cdots, vc_n\}$ denotes the set of dedicated communication channels transporting messages between tiles. If $\mathcal{P}_O$ is the set of all output ports of all communication interfaces of all tiles ($\bigcup_I \mathcal{O}$) and $\mathcal{P}_A$ is the set of all input ports of all tiles ($\bigcup_I \mathcal{A}$), then $vc_j = (p_{o_j}, m, p_{i_j})$ represents an uni-directional virtual channel which transports a message of unique type ($m \subseteq \mathcal{M}$) between sender port $p_{o_j} \in \mathcal{P}_O$ to a receiver port $p_{i_j} \in \mathcal{P}_I$ throughout the time-triggered communication subsystem $\mathcal{CS}$.

At the current version of the communication implementation, there is only `send_message` task. `receive_message` is implicitly done when a functional task starts. In other words, it begins with reading its memory-mapped I/O. The `send_message` is scheduled right after the end time of a functional task, which means the release time of the task added to the WCET.

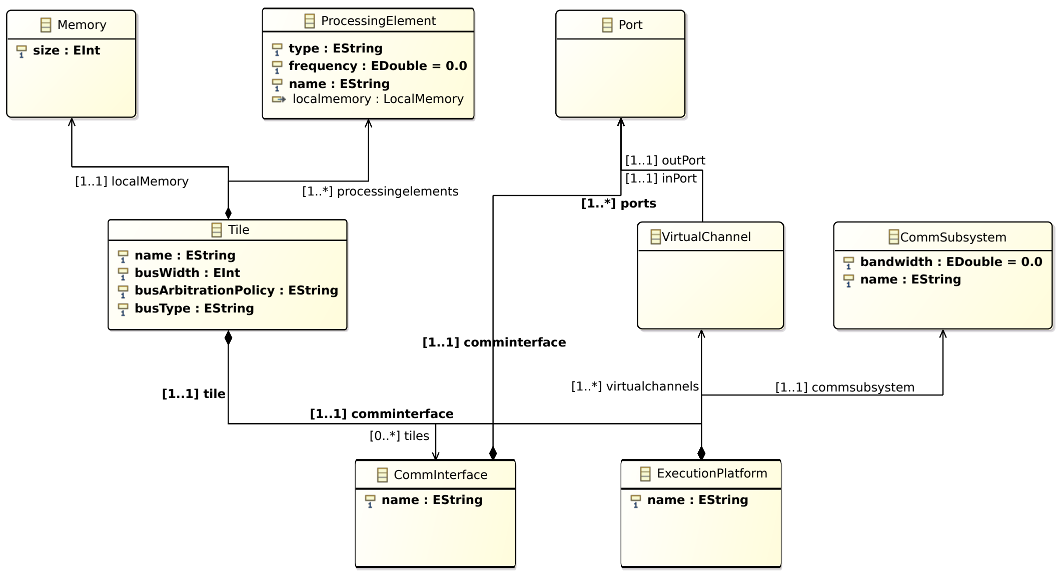


Figure 7.4.: UML diagram of Platform

7.3 Modeling the Mapping

The system mapping includes the processes of binding and scheduling the application model onto the resources of the platform model. It is the transformation of a directed graph of tasks' dependencies (Figure 7.1-a) into the task and message schedules (Figure 7.1-c).

If $\mathcal{K}$ is the set of tasks of all applications $\mathcal{A}$, $\mathcal{C}$ the set of all channels, and $\mathcal{T}$ the set of all tiles, and $\mathcal{VC}$ is the set of virtual channels configured at the hardware design time, then a mapping can be defined as a tuple $M = (\alpha, \beta)$ with:

1. The function $\alpha : \mathcal{K} \rightarrow \mathcal{T}$ maps every task to a unique tile (multiple tasks can be assigned to one tile).
2. The function $\beta : \mathcal{C} \rightarrow \begin{cases} \mathcal{VC} \rightarrow \mathcal{CS} \\ \mathcal{T} \end{cases}$ maps every channel either to a communication subsystem (which results in the creation of a dedicated physical channels for the virtual channels realizing the transfer of messages between two tasks on two different tiles), or to a private memory of tile (in the case where both communicating tasks are mapped on the same tile).

In the implementation phase, the mapping, as modeled in Figure 7.5, consists of four steps:

1. `Task-on-Tile` assigns each task to a unique tile.

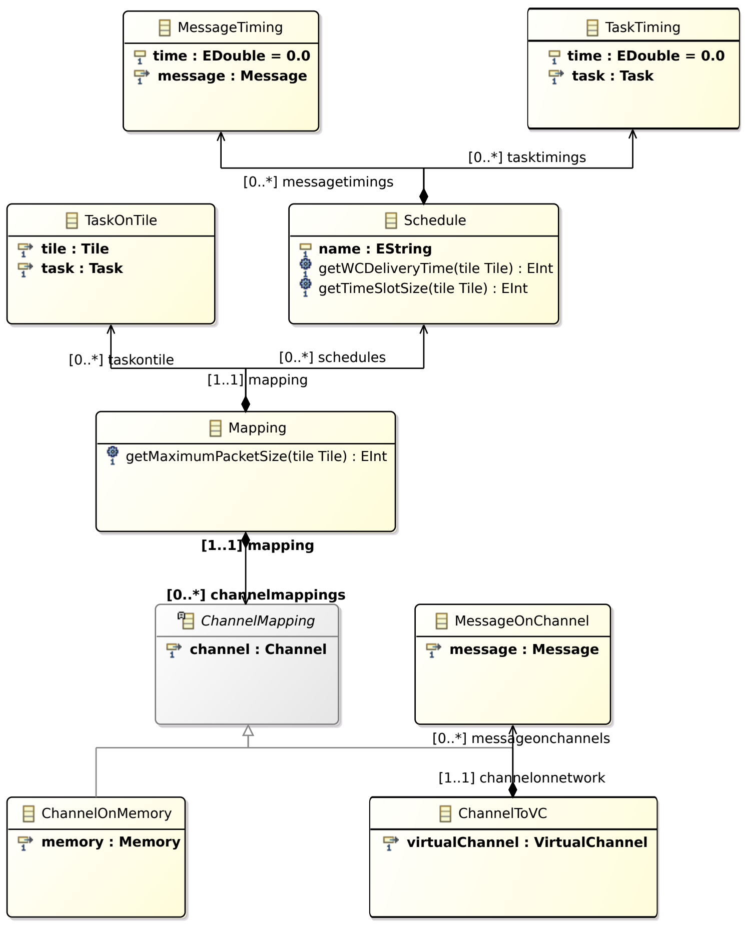


Figure 7.5.: UML diagram of Mapping

2. `Channel-on-Network` specifies the channel and the virtual channel that messages are mapped onto.

3. `Message-on-Channel` maps messages to channels in the NoC, which results in the creation of a dedicated VC, realizing the transfer of messages between two tasks on two different tiles.

4. `Channel-on-Memory` maps messages to a private memory of tile for inter-tile communication.

A `Schedule` for each task, and each message is defined as a consecutive set of time slots consisting of start and end time tuples. The schedule is unrolled to a hyperperiod which is a cyclic schedule.

Let $\mathcal{S}$ be the set comprising all starting points of all tasks, $\mathcal{M}$ the set of all messages transferred between all `Tasks` and SL_{comp} the set comprising all time slots of all

`Tasks` and SL_{com} the set containing all time-triggered communication time slots in the considered platform.

A TT-schedule δ_{PE} for each processing element is defined as a consecutive set of time slots $\{sl_0, ..., sl_n\} \in SL_{comp}$ consisting of start and end time tuples $sl_i = (sl_{i_{start}}, sl_{i_{end}})$ with a consecutive order $\forall i < n : sl_{i_{end}} \leq sl_{i+1_{start}}$. Now δ_{PE} maps each task of the specific PE onto a computation time slot of the TT-schedule of this PE:

$$\delta_{PE} : \{k_i \rightarrow sl_j | \forall k_i \in \mathcal{K} \downarrow_{\alpha(\mathrm{PE})}\}$$

where $\mathcal{K} \downarrow_{\alpha(\mathrm{PE})}$ restricts the number of possible tasks to the tasks that have been mapped on this PE.

A TT-schedule for each communication interface δ_I is defined as a consecutive set of time slots $\{sl_0, ..., sl_n\} \in SL_{com}$ consisting of start/end time tuples $sl_i = (sl_{i_{start}}, sl_{i_{end}})$ with a consecutive order $\forall i < n : sl_{i_{end}} \leq sl_{i+1_{start}}$. Now δ_I maps each virtual channel that has its output port in the specific interface onto a communication time slot:

$$\delta_I : \{vc_i \rightarrow sl_j | \forall vc_i \subseteq \mathcal{VC} : vc_i.p_o \in I.\mathcal{O}\}$$

In addition, we assume that if a dependency relation between two tasks $k_i, k_o \in \mathcal{K}$ exists, such that k_o is sender and k_i is receiver, then the receiver never starts before the sender has finished execution:

$$\forall (k_o, k_i \in \mathcal{K}) : \delta_{PE}(k_i).sl_{start} > \delta_{PE}(k_o).sl_{start} + k_o.\tau$$

GALI Simulation 8

Electronic Design Automation (EDA) has evolved during the past years towards more abstraction at a higher level. Design automation has started from low-level placement and routing automating in the Inintegrated Circuit (IC) design to numerous high-level system-level tasks. The EDA community now deals with developing tools and methods for the design of Cyber-Physical System (CPS).

In the design of CPS, all the system elements such as physical processors, control algorithms, and the base computation and communication platforms – on which the rest of the system is implemented – are tightly connected. Unfortunately, available EDA tools cannot handle such design and modeling. The missing part is the link between modeling tools like Matlab/Simulink used to model the system and tools used to configure the hardware/software platforms.

In the embedded system design, unlike general-purpose computing, dividing the design into layers to deal with individually is not easy. In the general case, each layer needs a properly designed interface to allow the different partitions to work together. A specialized application like embedded systems or CPS which involves hardware/software co-design, which are used in the automotive industry, for instance, poses serious problems when different groups design it with entirely different sets of expertise.

CPS needs an integrated modeling and design environment. In the current approach, each layer/partition of the design has an idealistic assumption (free of any errors) about the rest of the system. It is not realistic, i.e., there is no delay, no conflict in the design, etc. However, as the implementation platform becomes more complex and distributed, these assumptions do not hold anymore.

As CPS becomes more complex and distributed, the integration of such a system becomes more challenging, which often takes more than 50% of the design effort. Certification for safety-critical systems can also add extra time and challenge to it. Therefore, to address the incompatibility between different parts of the design, we need a framework to integrate all levels of abstraction. The assumption in each level or part of the design is fully compatible with the rest of the system.

To address these problems, there is a need for a EDA methods and tools that bridge the gap between the result of modeling and implementation of the hardware/software platforms. In this work, we have developed a framework to integrate the design and implementation of the time-triggered systems.

The first section of this chapter describes how IA simulation technology works. Next, our proposed simulation technique is presented, which uses the nature of time-triggered architecture combined with the IA logic to speed up the simulation. It also describes how the scheduling procedure of our solution works. Finally, we conclude this chapter by presenting our framework called GALI-CE.

8.1 IA Simulation Technology

In Section 2.5, we mentioned the instruction accurate (IA) simulator. In this section, we explain how its internal logic works and how one can tune an IA simulator to speed up the simulation of a time-triggered system.

In available instruction accurate simulators, a very basic timing model is included. The processor model counts the number of executed instructions, and based on the annotated nominal processor speed in MIPS, it derives the simulation time. To change the time granularity of IA simulators, one can set different simulation steps, called *quantum size* or, in other words, the context switch allowance. In a multiprocessor simulation, a quantum is the number of instructions each processor executes in each turn before the simulation engine switches to the next processor. At the end of each round, when all processors have completed their quantum, the simulator synchronizes the simulation result and the simulation global time. The order in which the processors are executed in each round is not deterministic.

The quantum value and MIPS rate are set by the user based on the trade-off between accuracy and simulation speed. Increasing the number of instructions running on a processor in one quantum will reduce the number of context switching and synchronization, improving simulation speed. However, it will increase the chance that synchronization causes gross functional errors, and interactions between processes become inaccurate concerning the timing, primarily if they communicate through a shared memory [42].

Simulating a safety-critical time-triggered system with available IA simulators might corrupt the time-triggered event occurrence time if a user chooses an unsuitable value for the quantum size and/or MIPS rate. To circumvent this, the quantum value must be selected so that the events occur precisely at the beginning of the simulation steps. Otherwise, the quantum duration will shift the event time (all events belonging to a quantum have the same occurrence time). Furthermore, since all the WCET values are pessimistic, tasks usually have slack at the end of execution before the next task. Therefore, a wait function should delay the simulator and enforce the time-triggered behavior. This is possible in IA simulators but cannot be generalized for each event. The scheduler should execute the events concerning the event's exact release time.

Besides the problem with setting the correct value for MIPS and quantum, instruction-accurate simulators abstract hardware implementation details (e.g., cache, pipeline, etc.) to increase the simulation speed. This abstraction causes inaccuracy in the simulation's timing information, i.e., the simulation of a function might take more or less time than what it takes in the real world. The hardware abstraction and MIPS and quantum value all affect the simulation and interaction between tasks in the simulation.

To solve this problem of simulating the safety-critical time-triggered system with IA simulators, the simulation engine should trigger the tasks in the foreseen scheduled order and execute the next task after finishing the dependent tasks. In other words, instead of just running instructions until all instructions are completed, the scheduler must start functions strictly at the corresponding point in time and switch to the next task at the right moment.

In the following, we propose GALI scheduler to handle the issues in an Instruction Accurate virtual platforms for time-triggered system.

8.2 GALI Technique

The technique developed in this work is based on an underlying IA simulator, which does not need to provide a timing model for fast pure functional testing. Instead, timing information from the static time-triggered configuration is linked to the functional model, adding the necessary timing information to the functional and instruction-accurate simulation. This model runs on the exact timing granularity as high-level models but executes the same binary code that will be performed on the actual processor or MPSoC platform, with complete functional observation and debugging capabilities.

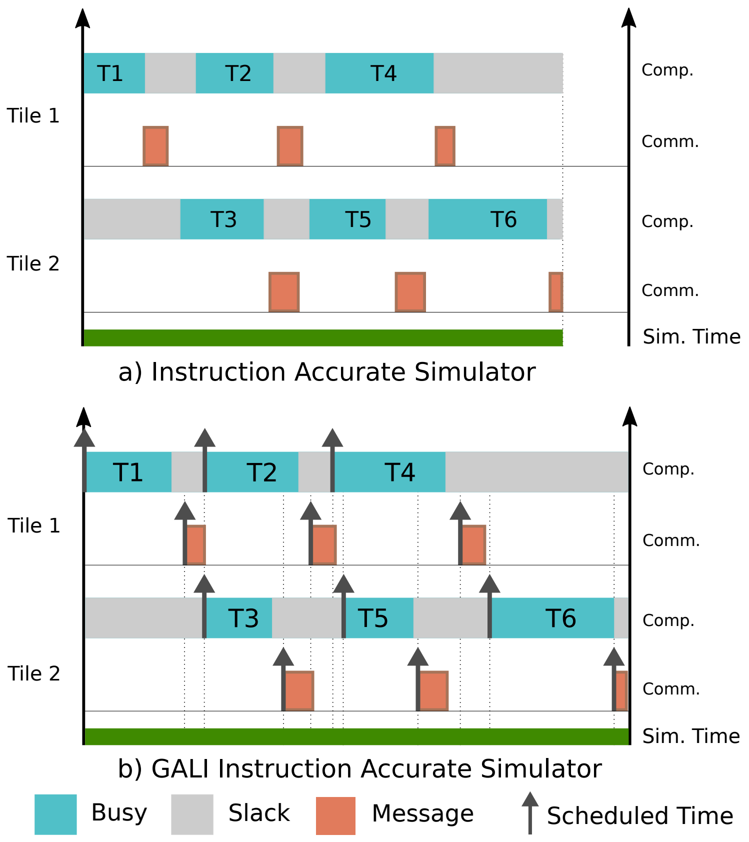


Figure 8.1.: Instruction Accurate simulation timing deviation versus GALI simulation

GALI is a simulation scheduling model based on an IA simulator that executes a predetermined time-triggered system configuration. The simulation model is called *Globally Accurate Locally Inaccurate* (GALI) because the order and time of task execution are accurate concerning the predetermined schedule (Globally Accurate). However, the timing behavior within a task is not accurately represented (Locally Inaccurate).

Figure 8.1 depicts an imaginary time-triggered system simulation scenario, first performed by a standard instruction-accurate (IA) simulator (Figure 8.1-a), and the second one with an instruction-accurate simulator equipped with the GALI scheduler. Figure 8.1-b shows precisely how the system should execute and when the communication points should occur at the scheduled time. However, Figure 8.1-a illustrates a timing deviation in executing tasks (due to, for instance, the abstraction of the simulation model) that affects subsequent tasks and communications, violating the execution behavior. Figure 8.1-a shows that the simulation time is less than the actual scheduled simulation time (tasks are completed earlier than expected). Even worse, the simulation is not completed within the timing window.

To address the issues illustrated in Figure 8.1-a, we employ the GALI scheduling algorithm, which is capable of providing accurate execution timing for time-triggered systems. The GALI algorithm (as described in Algorithm 8.2.1) takes as input a list of processors, where each processor has a list of tasks and messages. Each task and message has a start time, a duration (WCET), a processor index, and a period which is the frequency or how often a task should be executed. These values are pre-computed based on analytical approaches. The algorithm then calculates the hyper-period, which is the least common multiple of the periods of all tasks and messages for all processors. The hyper-period is the main global schedule that is going to be repeated throughout the system lifetime.

Next, the algorithm populates the hyper-period by adding the tasks and messages as events into a sorted list. An event is a tuple of execution (aka running) time and processor index. The execution time is equal to the start time plus the duration (WCET). The hyper-period list is sorted based on the execution time.

To populate the hyper-period, the algorithm iterates through each processor and, for each task or message in the processor's list of tasks and messages, it computes the number of repetitions of the task or message during the hyper-period. For each repetition, it computes the start time and execution time of the task or message, and adds the tuple [execution time, processor index] to the event set.

Finally, the algorithm outputs the event set, which contains all the tuples indicating the execution time and processor index for all tasks and messages that should be executed during the hyper-period.

GALI repeats the Hyperperiod until the simulation is finished or interrupted. During each iteration of a GALI implementation of the time-triggered system, the following steps occur:

1. *Interception*: Events are labeled in the application programs. The scheduler operates as the main state machine, and each state of the central state machine is a resource with events (Processing Elements or Network Interfaces). Each state machine of the parent resource keeps track of its events. The interception library, a debug interface that is sensitive to our defined labels, captures events during the simulation and signals the GALI scheduler to change the state accordingly.

2. *Event Capturing*: When GALI moves to the next state, both the main state machine and the state machine of the event's owner change. Transitions occur at the boundaries of events that are fixed points, similar to cooperative multitasking. The system is partitioned, and no inter-communication occurs

Algorithm 8.2.1 GALI TT Scheduling Algorithm

Let:

n be the number of processors
m be the number of Tasks and Messages for each processor
$P(i, j)$ be the Period of the j-th task or message for the i-th processor
$S(i, j)$ be the Start time of the j-th task or message for the i-th processor
$D(i, j)$ be the Duration (WCET) of the j-th task or message for the i-th processor
H be the hyperperiod
$E(k)$ be the k-th event in the hyperperiod, where $1 \leq k \leq$ total number of events

```
procedure SCHEDULER
    Calculation of hyperperiod
    $H = LCM_{i=1,j=1}^{n,m}(p(i,j))$                    ▷ LCM of period of all elements
    Populate list of events
    for $i \leftarrow 1$ to $n$ do                               ▷ Processor index
        for $j \leftarrow 1$ to $m$ do                          ▷ Task/Message index
            for $k \leftarrow 1$ to $ceil(H/P(i,j))$ do   ▷ number of repetitions during H
                $ExecutionTime \leftarrow (k-1) * P(i,j) + S(i,j) + D(i,j)$
                $E \leftarrow E \cup (ExecutionTime, i)$
            end for
        end for
    end for
    $E \leftarrow Sort(e \in E)$                        ▷ Sort based on $ExecutionTime$
end procedure

procedure EXECUTION
    while $True$ do
        for $e$ in $E$ do
            $e$.execute
        end for
    end while
end procedure
```

within an event between two processes. If any communication needs to happen, it is an event and does not corrupt the system. Therefore, processors can execute independently, as long as the order of event execution is preserved.

3. *IA Functional Simulation*: The simulator runs the same binary on the hardware and provides a functional testing environment to give the user the desired debuggability and observability of the system.

As described above, the simulator is globally accurate concerning the time-triggered events, and the result of the system simulation complies with the system schedule. However, it is locally inaccurate because the time within two consecutive events is not accurately known in the simulation and the application running on top of it. In other words, we only take care of the executed number of simulated (target) processor instructions. We do not consider the time the processor has consumed to execute

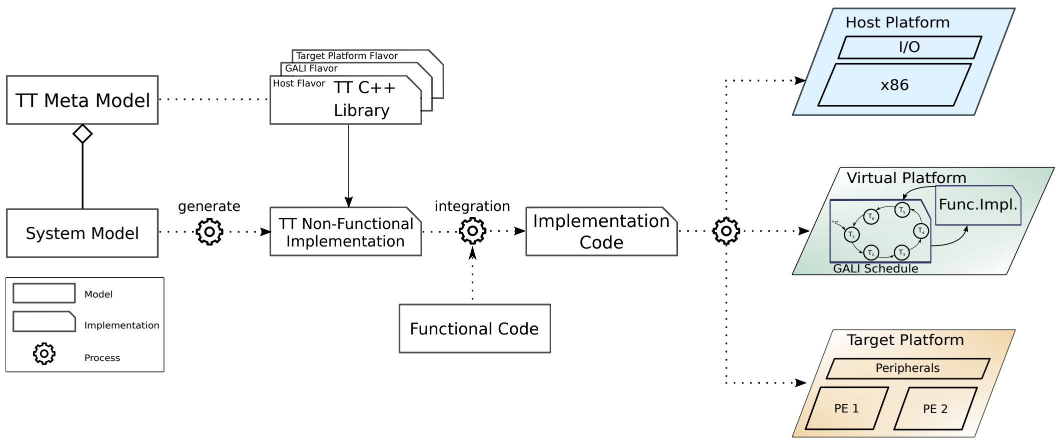


Figure 8.2.: Overview of the Framework

these instructions. We only maintain and (re-)construct the global and observable time-triggered system events. The GALI scheduling technique is independent of a specific simulator platform or vendor, and any instruction or cycle-accurate simulator that can execute a time-triggered system can be used.

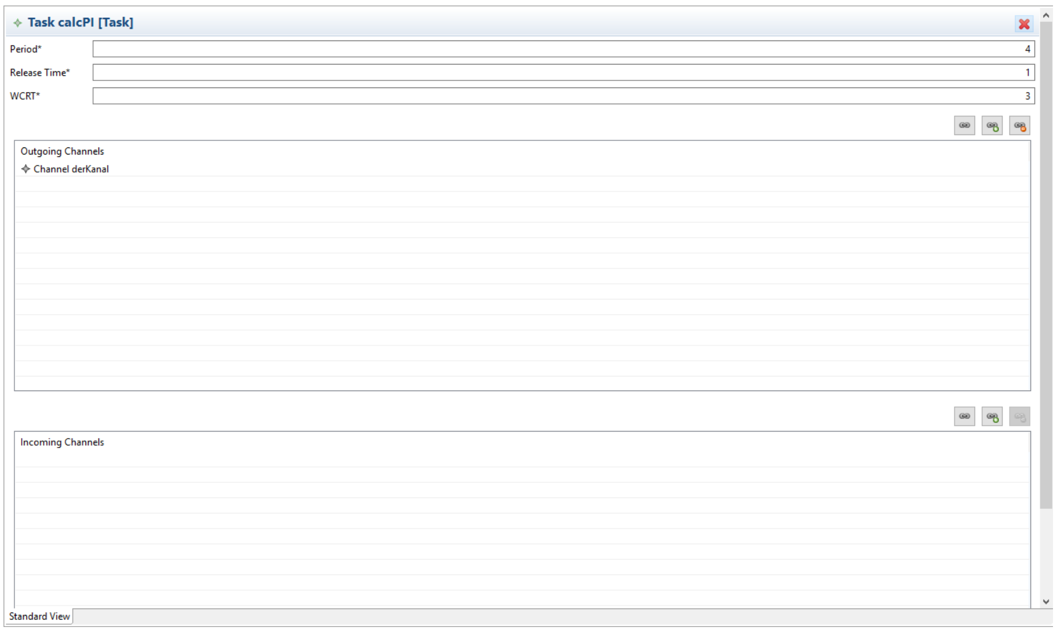
8.3 GALI Configuration Environment

The purpose of the GALI Configuration Environment (*GALI-CE*) is to facilitate the generation and testing of the target time-triggered system concerning the predefined schedule and functionality. A configuration framework is a tool to instantiate a time-triggered system from the provided meta-model and generate compatible binaries for verification and validation purposes for different use-cases.

It simplifies the complexity of implementing a time-triggered system by abstracting all the error-prone configurations from the designer. Figure 8.2 shows the design flow that is described in the following. First, we represent the system model creation and generation, then different flavors of the output are explained. We conclude this chapter with how GALI-CE is used as a verification tool.

8.3.1 System Model Creation

As shown in Figure 8.2, the system model is created based on the GALI-CE meta-model. The GALI-CE frontend represents the meta-model of the time-triggered system, which the designer can instantiate. The frontend provides the user interface



Task calcPI [Task]

Period*	4
Release Time*	1
WCRT*	3

Outgoing Channels

Channel derKanal

Incoming Channels

Standard View

Figure 8.3.: GALI-CE GUI - Define Task Interface

to make the system and the application definitions to suit the meta-model for a given use case, e.g., a vehicle control unit or a flight controller.

The GALI-CE frontend is implemented in Java and is based on the Eclipse framework. It uses the Eclipse Modeling Framework (EMF) as the data backend and Eclipse EMF Client Platform (ECP) for the data representation of the user interface. The EMF model is designed using the UML diagram in the Eclipse Papyrus modeling environment. The UML model is converted to a meta-model (Ecore) for describing models with auto-generated model code. Figure 8.3, just as an example of the graphical user interface, gives a glimpse of the interface for defining tasks.

8.3.2 System Model Generation

The GALI-CE time-triggered C++ library provides Application Programming Interface (API) for the implementation of the meta-model. It defines the platform, tiles, tasks, and communication channels. The API helps to define the system. `Platform` API defines the whole system. All other components are bound to the platform. `Tiles` represents the tiles or partitions. `Tasks` classes inherit from the `task_base` class and the instances should implement the `do_main()` method which is the functionality of the task. The front end generates a configuration code based on the user's model. The main function instantiates all the necessary elements to make the schedule and map different system elements.

GALI-CE frontend generates C++ task template to be extended by the user's function code. It is called non-functional implementation because it is a placeholder for the user code. It consists of C++ files and needed header files to implement the tasks. The generated C++ files implement the `task_base.hpp` virtual methods.

8.3.3 Output Flavors

The GALI-CE has three different flavors; host simulation (native), IA simulation with GALI, and target platform flavor. Different sets of outputs will be generated depending on the designer's flavor.

- The *native* flavor simulates the system natively on the host machine, and no instruction set simulator is necessary. Native compilation helps run the system as a generic system by abstracting the low-level drivers.

- The *GALI* flavor cross compiles the implementation code to the bounded PE. It also generates a state machine based on the hyperperiod, which acts as the custom scheduler for the virtual platform (VP). It also generates the scheduler configuration for the VP. The configuration tells the VP custom scheduler what should be simulated next and sets the corresponding simulation time. It is a tuple consisting of tile identifier (ID) or core ID, and WCET. The custom scheduler performs the simulation concerning this scheduling array of tuples. This is used in the "harness" file of the virtual platform to orchestrate the order of the simulation. A *harness* is a C/program that makes calls into the simulator API to connect and control the simulation component. This file is linked to the simulator to provide a binary. The `main()` function definition includes in this file and includes a command-line parser to parse all the arguments passed to the simulator.

- The *target platform* flavor generates the same cross-compiled binary as for GALI. The difference between these two flavors is the label codes that are embedded in GALI flavor, which tells the virtual platform when to switch to another task. Instead of a virtual platform configuration file, the target platform flavor generates a Xilinx Vivado system definition. It generates a TCL script that describes the system block design, which can be used by the Xilinx Vivado tool along with the necessarily Intellectual Propertys (IPs) to generate the system and the bitstream.

8.3.4 Verification

Besides the items mentioned above, GALI-CE is also a verification tool that checks if the schedule is feasible and the task definitions and timing features that the designer provides are consistent, and no conflict exists. To do so, it can use SDF or UPPAAL verification language for the timed-automaton formalism [49].

To verify the system's correctness, one should check if the functional code executes the static schedule the same as the analytical model implementation. This can be online, meaning it is running on the system, or offline by comparing the execution trace.

For the online verification or monitoring, GALI-CE can be extended to automatically generate an "Observer" from the timed-automata specifications of the UPPAAL and to check during the runtime if, while the system is running, the function execution matches the generated timed-automata.

For the offline verification, GALI-CE compares the trace generated by the execution of the code on the system with the golden model generated by the analytical model of SDF tool. If the recorded trace does not match the specification, the user can fix the flaw in the design in the early stage. Trace checking complements the verification performed at the design time (e.g., Correct-by-Construction) before deploying on the target platform.

9 Target Implementation

In this section, we apply the approach from Section 8 to our Multirotor use case and present how we can define the platform and tasks, configure the target design, verify and validate the generated system, and perform the test on the implemented design which is the goal of the Section 10.

Items that are explained in this chapter are products of *GALI-CE* toolchain. It provides an API for a time-triggered C++ library to define platform, tiles, tasks, and communication channels. All the necessary supporting files will be generated based on the target platform.

Figure 9.1 shows an overview of the GALI-CE toolchain when applied to our multi-rotor use case. On the one hand, it deals with the actual implementation for both HIL and VPIL setups, such as a set of files to be completed for the implementation of the system, a global GALI scheduler for the simulator, and the complete Vivado project as well as all necessary binaries.

On the other hand, it supports verifying the timing requirement for the configuration and validating and verifying the generated system. In the following, we describe different parts of our configuration environment.

9.1 Instantiations & Definitions

As mentioned in Section 8.3, GALI-CE frontend offers a Graphical User Interface (GUI) based on the Eclipse Modeling Framework (EMF) to define and model the system. The front end generates all the necessary files and placeholders to be extended by the user.

The front end asks for the task's name, start time, associated processor, and period. It generates a flattened hyper-period as a list to be executed infinitely by the processor. Listing 9.1 shows the generated tasks list (tuple of `(timestamp, task_function)`) associated with the flight processing unit consisting of the following tasks: `Task_sensor_processing`, `Task_remote_processing`, `Task_controller`.

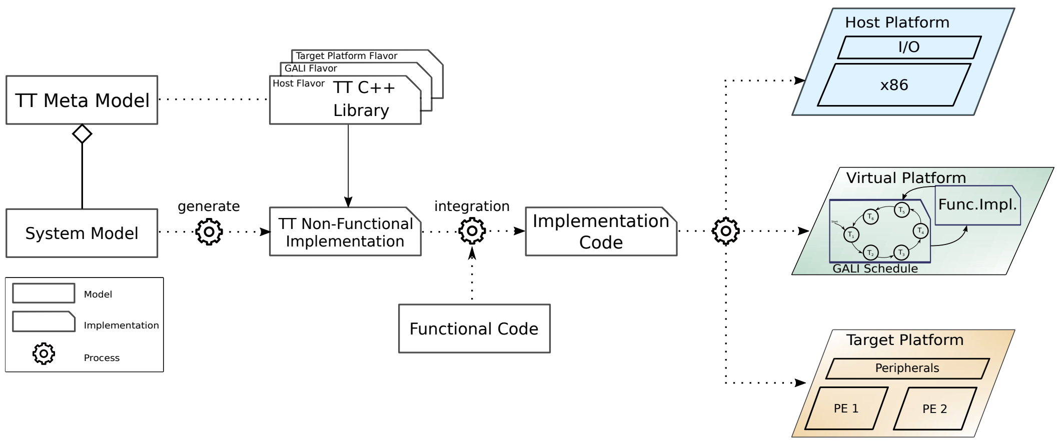


Figure 9.1.: Experimental Flow
Trace of the simulation is compared against the trace of timed automata representation of timing requirement to validate the output

```
task_execution tasks[] = {
// Period 1
  {601, &Task_sensor_processing}, {676, &Task_remote_processing},
  {695, &Task_controller},
// Period 2
  {2601, &Task_sensor_processing}, {2676, &Task_remote_processing},
  {2695, &Task_controller},
    ...
// Period 10
  {18601, &Task_sensor_processing},{18676,&Task_remote_processing},
  {18695, &Task_controller}
};
```

Listing 9.1: Flight Processing Task Schedule

The same task schedule (tuple of (`timestamp, task_function`)) is generated for the sensor processing unit (Listing 9.2) with the following tasks: `Task_read_imu`, `Task_start_mag`, `Task_voting`, `Task_write_motors`, `Task_read_bu`, `Task_read_mag`, `Task_read_remote_ctrl` (`Task_voting` is not implemented, we kept it for backward compatibility with the previous implementation with triple-modular redundancy).

For each task mentioned in the task list, there are also a header file `.h` and implementation file `.cpp` generated that the system designer should complete. Listing 9.3 shows one example of a task function (here is `Task_sensor_processing`) generated by the tool.

```
task_execution tasks[] = {
// Period 1
  {0, &Task_read_imu}, {601, &Task_start_mag},
  {1180, &Task_voting}, {1220, &Task_write_motors},
// Period 2
  {2000, &Task_read_imu}, {3180, &Task_voting},
  {3220, &Task_write_motors},
    ...
// Period 10
  {18000, &Task_read_imu}, {19180, &Task_voting},
  {19220, &Task_write_motors}
};
```

Listing 9.2: Sensor Actuator Unit Task Schedule

Tasks classes inherit from the `task_base` class and the instances should implement the `do_main()` method which is the functionality of the task.

```
/*
 * task_sensor_processing.cpp
 * Auto generated by the Frontend
 */
#include "task_sensor_processing.h"
platform_error Task_sensor_processing::do_main()
{
  receive_sensor_data_message(&sensor_input_data);

  /* TODO Add the functional code */

  return platform_error::PLATFORM_NO_ERROR;
}
```

Listing 9.3: Auto-generated template for Sensor Processing Function

Next to tasks, GALI-CE generates a schedule for messages as well. On the hardware implementation, the primary function uses this list of `timestamp` and programs the message timer associated with each processing unit. Listing 9.4 shows the schedule for sending messages on the sensor-actuator unit.

```
task_message_set taskMessageSets[] = {
  {541},   // For period 10
  {2541},  // For period 1
  {4541},  // For period 2
  ...
  {16541}, // For period 8
  {18541}  // For period 9
};
```

Listing 9.4: Message Schedule For Sensor Processing Unit

To have a better understanding of how all these tasks, timers, and lists work together, Listing 9.5 shows the primary function of the processing unit. It is the same for both flight controller and sensor-actuator units.

```c
#include "xparameters.h"
#include "network_interface.h"
#include "app_def/app.h"
#include "xtmrctr.h"
#include "xintc.h"
#include "drivers/timers/task_timers.h"
#include "drivers/timers/message_timers.h"

int currentTaskIndex = 0;
task_execution* currentTask = 0;
int currentMessageIndex = 0;
task_message_set* currentMessage = 0;
int newTask = FALSE;
XIntc InterruptController;

void task_timer_interrupt() {
  currentTask = &tasks[currentTaskIndex];
  currentTaskIndex++;
  if (currentTaskIndex >= TASKS_SIZE)
    currentTaskIndex = 0;
  newTask = TRUE;
}
void message_timer_interrupt() {
  currentMessage = &taskMessageSets[currentMessageIndex];
  currentMessageIndex++;
  if (currentMessageIndex >= MESSAGES_SIZE)
    currentMessageIndex = 0;
  programMessageTimer(taskMessageSets[currentMessageIndex].sendTime);
}
int main() {
  initHardware();

  if (tasks[currentTaskIndex].startTime > 0) {
    programTaskTimer(tasks[currentTaskIndex].startTime);
  } else {
    task_timer_interrupt();
  }

  if(MESSAGES_SIZE > 0)
    programMessageTimer(taskMessageSets[currentMessageIndex].sendTime
   );

  for (;;) {
    if (newTask) {
      newTask = FALSE;
      programTaskTimer(tasks[currentTaskIndex].startTime);
      (*currentTask).do_main();
    }
  }
  return 0;
}
```

Listing 9.5: Main Function of Processing Element

Another important output of the GALI-CE is the global GALI scheduler. Assuming SP representing Sensor Processing (ID = 0) and FC for Flight Controller (ID = 1) processing elements, Listing 9.6 depicts the tuple `(ProcdID, timestamp)`. It is a flat schedule built from the hyper-period. Therefore it has timestamps for ten periods because the hyper-period, in this case, is made out of 10 basic periods.

```
#define SP  0 // Sensor Processing Processor
#define FC  1 // Flight Controller Processor

struct Schedule {
    int ProcId;
    int TimeStamp;
};

// Number of events in a schedule
const Uns32 NUM_EVENT = 65;
const Uns32 HYPER_PERIOD = 20000;

struct Schedule Schedules[] = {
/* ProcessorID  TimeStamp */
// Period 1
  { SP,    0 }, // Task_read_imu
  { SP,    601 }, // Task_start_mag
  { FC,    601 }, // Task_sensor_processing
  { FC,    676 }, // Task_remote_processing
  { FC,    750 }, // Task_controller
  { SP,    1180  }, // Task_voting
  { SP,    1220  }, // Task_write_motors
```

Listing 9.6: GALI Global Task Schedule

9.2 Verification

GALI-CE can check the feasibility of the input schedule and compare traces for verification and validation of the implemented system. We use Timed Automata (TA) to capture the behavior of the time-triggered scheduler and to analyze the actual timing progress and activation instants of tasks.

Timed automata and model-checkers like UPPAAL[49] are very suitable for capturing and verifying the behavior of time-triggered systems since concurrency and synchronization are modeled explicitly. The timed automaton directly implements the timing requirement of the input schedule. We derive a functional trace by the UPPAAL tool, which gives the activation instants of the tasks within a hyperperiod and their delay time (annotated WCETs). We then use this trace as a reference trace to check the validity of activation instants in the instruction accurate simulation trace (see Figure 9.1-UPPAAL for trace verification tool).

For simplicity of presentations, the notation used for the UPPAAL model combines T with the task number, e.g., $T1$, which represents `Task_read_imu` which is used in our C++ implementation. Table 9.1 provides a mapping of the timed-automata locations and different tasks. All the tasks in the multirotor functionality are shown together with their WCET in Table 9.1. Task, $T1$ to $T8$, belong to the sensor actuator unit, and $T9$ to $T11$ are the controller's duty. The next column in the table shows the frequency of execution of each task in the hyper-period, and the last column depicts how long each task would take to finish in the worst case.

Figure 9.2 depicts different tasks of the Table 9.1 from the time-triggered schedules, including Microblaze$_0$ (MB0), Microblaze$_1$ (MB1), and the communication infrastructure of MB0 (consisting of VC1 and VC3) and MB1 (consisting of VC2). In the Figure 9.2, the activation time of tasks and their maximum duration (WCET) are expressed by the clock variable t. For instance, $T1$ is activated at time $0ms$ and is finished at time $55ms$.

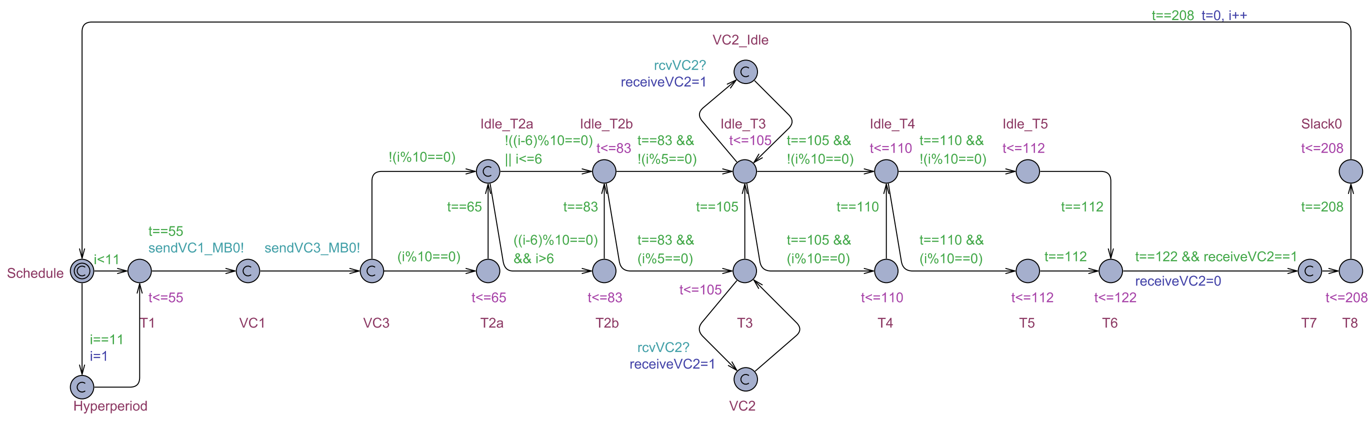
There is also conditional activation of tasks for tasks with less execution frequency. It means that they are not executed in every period, and therefore they have additional guard conditions on the hyper-period counter variable i. For example, while $T1$ is activated in every period, $T2a$ is activated each 10*th* period. The tasks that will not be executed in the current period are so-called idle. They are denoted in Figure 9.2 as *Idle_*Tx and are expressed with a start and an end time.

All the time slots dedicated to each task are based on the WCET, which is the maximum needed time to finish the execution of the task. Since most of the tasks do not use all the time slots, we will have slack time (idle CPU), which is denoted by *Slack*x and is expressed with a start and an end time.

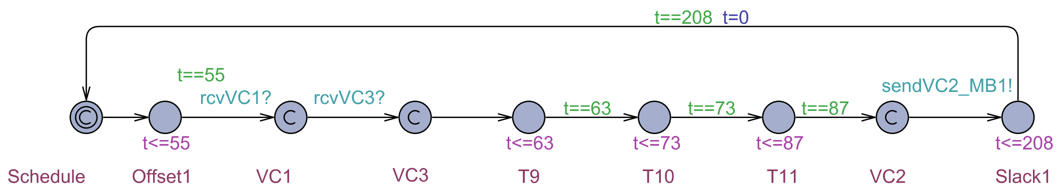
TA location	Function/Task	Period [ms]	WCET [ms]
T1	Read Gyroscope & Accelerometers	2	0.541
T2a	Read Magnetometer (start measurement)	20	0.091
T2b	Read Magnetometer (read sensor data)	20	0.273
T3	Read Barometer	10	0.212
T4	Read Remote Control	20	0.049
T5	Read Battery Guards and Temperature	20	0.011
T6	Write Remote Control Display	2	0.100
T7	Voting (not used here)	2	0.100
T8	Set Motor Values	2	0.851
T9	Sensor Processing	2	0.075
T10	Remote Processing	2	0.019
T11	Flight Controller	2	0.035

Table 9.1.: Multirotor's Tasks

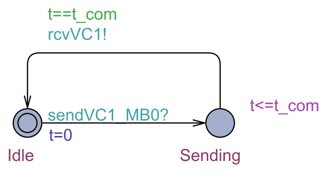
The described output generated from the UPPAAL model-checker in this section and the virtual platform are compared against each other in Section 11.3 which proves if the implementation satisfies the definition.



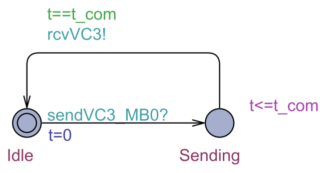
(a) Microblaze$_0$ (MB0) with start condition $t := 0$ and $i := 1$



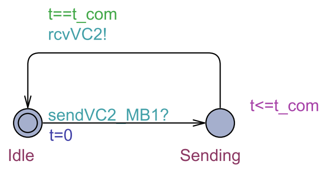
(b) Microblaze$_1$ (MB1) with start condition $t := 0$



(c) NoC VC1



(d) NoC VC3



(e) NoC VC2

Figure 9.2.: time-triggered schedules for used HW resources (expressed as timed automata)

Part III

Evaluation and Results

Overview 10

The evaluation sections are structured along with the contributions of this thesis. In this section, the definition of the evaluation goals is presented. Next, the setup needed to perform the evaluation and measurement infrastructure for both Hardware-in-the-Loop and Virtual Platform-in-the-Loop is explained. The results are then used to verify and validate the concept, tooling, and infrastructure. The evaluation section focuses on the GALI technique.

In Section 11.1, the Hardware-in-the-Loop experiment is performed to gather the golden result for the comparison of the accuracy of VPIL prototyping. Section 11.2 gives an overview of the accuracy of GALI in comparison to the different virtual platform scheduling against HIL as a golden result. Section 12 concludes this thesis by comparing different experimental results and comparing the accuracy and benefit of this thesis's contribution, i.e., GALI simulation technique, and GALI-CE framework.

10.1 Goals of the Evaluation

The evaluation has been done with different setups; Virtual Platform-in-the-Loop (VPIL) and Hardware-in-the-Loop (HIL). A flight simulation software has created a deterministic and reproducible simulation environment.

The main goal of the evaluation is to see, first of all, how accurate is the GALI model in comparison to the actual hardware. Second, we would like to know the accuracy of other simulation techniques. Furthermore, we are interested in figuring out the speed-up factor of the GALI when it is used instead of a standard instruction accurate (IA) simulator with or without timing models.

10.2 Evaluation Setup

This section gives an overview of all the elements used in our HIL and VPIL experiments. We explain the setups by introducing different ingredients used in other evaluation flows.

10.2.1 AeroSim Flight Simulator

AeroSIM RC[50] is a full-function Radio Control (RC) flight simulator system that is capable of providing a simulated environment to fly RC aircraft. AeroSIM RC offers a simulation of several types of RC model aircraft, including airplanes, helicopters, and multirotors such as a quadcopter and hexacopter.

The main feature that sets this simulator apart is the wide variety of configurable multirotor simulations. It supports research and development through third-party Dynamic-Link Library (DLL) which runs inside the simulation and allows for control of the model via software. OFFIS has developed a SimLink client DLL that exchanges the data between the AeroSIM RC flight simulator and the external remote controller. It converts the values received from the Ethernet to AeroSim-understandable commands.

AeroSIM RC lets the user record and replay the remote control commands for later playback. It helps us to be able to reproduce the same flight commands over and over for different settings. A screenshot of the simulation environment for a multirotor is shown in Figure 10.1.

10.2.2 Multirotor

A multirotor, in our case a quadcopter (a multirotor with four rotors) shown in Figure 10.2, is a rotorcraft that moves by the relative speed of each motor to the thrust and torque of each engine.

The rotors provide six degrees of freedom which refers to the freedom of movement in a $3D$ space as shown in Figure 10.3a. The multirotor can freely change position to forward/backward, up/down, and left/right axes. The orientation can also change through rotation about the three axes, yaw, pitch, and roll. These directions are considered in the design of our multirotor as illustrated in Figure 10.3b.

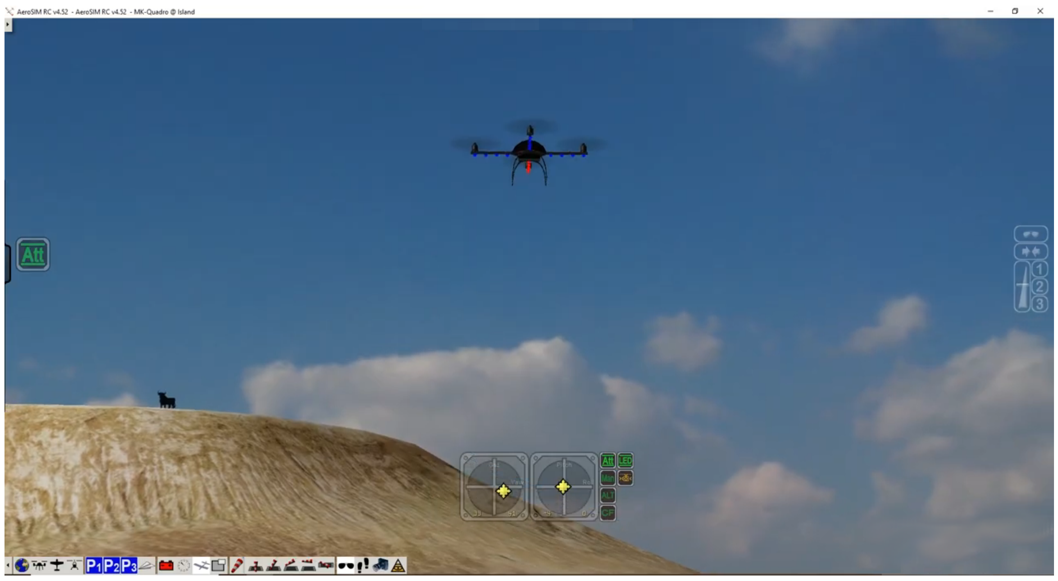


Figure 10.1.: Screenshot of AeroSIM RC multi-rotor simulation environment

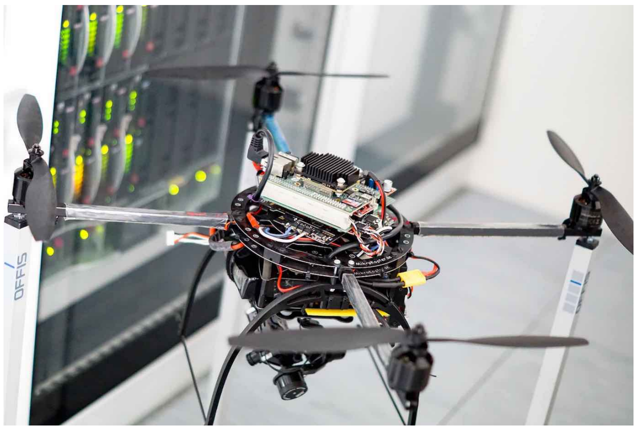
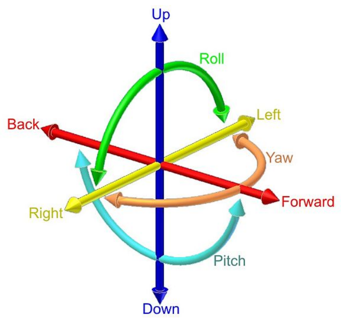
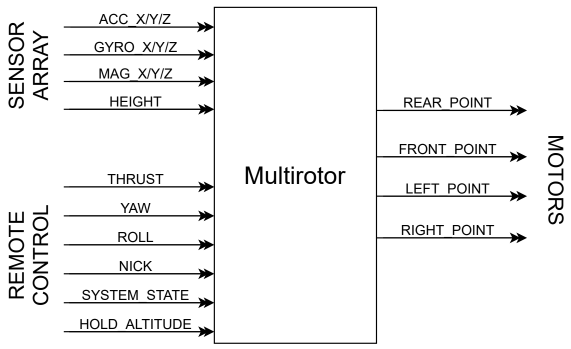


Figure 10.2.: Quadcopter developed in OFFIS [Copyright OFFIS]

In the controller of the multirotor, different variables impact the movement of the vehicle. Different sets of sensors give the controller the necessary knowledge about its environment. The pilot tells the controller hot to move the multirotor by changing the output of the remote controller. The controller is just a Proportional Integral Derivative (PID) that regulates the motor values based on the remote control signals and the environmental variables. Figure 10.3b shows all the mentioned inputs and outputs of the flight controller more specifically.

The system accesses the sensors and the motors through a 400kHz I^2C protocol and receives instructions from the remote through a 50Hz Pulse-Position Modulation (PPM) signal. From reading the sensors until returning the motor set points, the control loop is executed in 2ms at a frequency of 500Hz. Our simulation setup is,





(a) Six Degrees of freedom of a multirotor[1] (b) Input/Output of the Multirotor

Figure 10.3.: Qaudcopter overview of maneuver capabilities

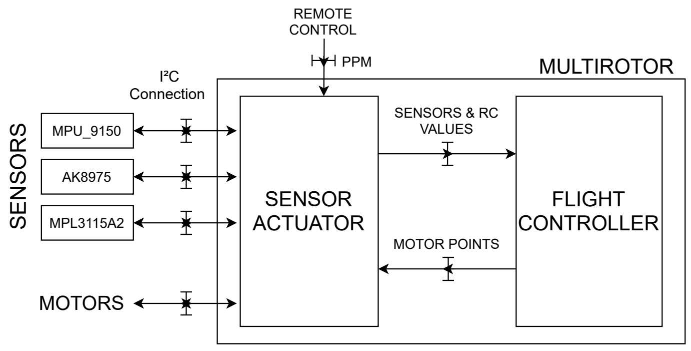


Figure 10.4.: Multirotor's architecture overview

however, slower. It takes in the worst-case 40ms until the adapter board sends a value and receives the answer from the PC. Therefore, we implemented a synchronization mechanism on the multirotor and adapter boards to block the system until we ensure the values are consistent.

The design decomposition of our multirotor is presented in Figure 10.4. The multirotor system consists of a flight controller and a sensor-actuator unit, each implemented on a separate Microblaze processor. They are communicating with each other via a communication interconnect that implements time-triggered communication through unidirectional VCs. The sensor processing is the I/O processing element and is connected to all external sensor and actuator interfaces. It transfers the processed flight data to the flight controller. The flight controller executes the main computation tasks, uses the sensor-actuator unit's data, and delivers the final motor set points. Figure 10.5 illustrates the building blocks of a processing unit and gives an overview of all the safety-critical functions used in the multirotor Figure 10.4.

[1]https://fr.wikipedia.org/wiki/Fichier:6DOF_en.jpg

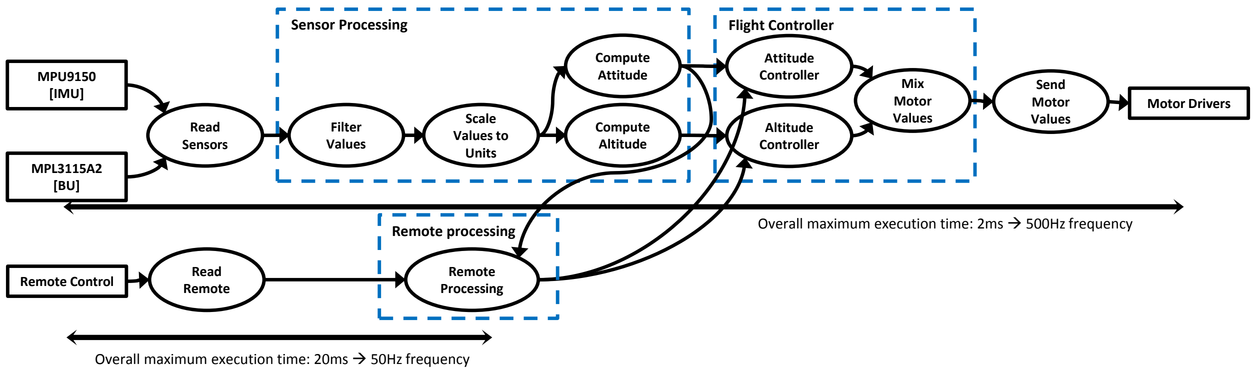


Figure 10.5.: Functional Units used in Multirotor [51]

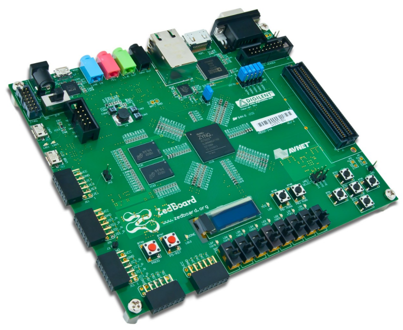


Figure 10.6.: Digilent Zedboard used for multirotor

The board chosen for it is ZedBoard Zynq-7000 ARM/Field-Programmable Gate Array (FPGA) System-on-a-Chip (SoC) development board as shown in Figure 10.6. The criterion for choosing this board is the capacity of chips concerning the design. ZedBoard is a Xilinx Zynq-7000 All Programmable SoC that contains all the necessary interfaces and supporting functions to enable a wide range of applications. Each of these processing units is mapped to a specific Microblaze processor on the FPGA. MicroBlaze$_0$ (MB0) is the sensor and actuator I/O processing element, and MicroBlaze$_1$ (MB1) executes the main computation tasks of the flight control application. Figure 10.7 depicts the mapping of the multirotor to the hardware platform.

MB0 is connected to all external sensor and actuator interfaces. Its primary responsibilities are executing the *Read Sensors*, *Read Remote*, and *Send Motor Values* task. The sensor and remote data are then transferred to MB1 via VC1. MB0 receives the computed motor set points via VC2 and transfers them to the motor drivers via an I^{2}C bus.

MB1 consists of *Sensor Processing*, *Remote Processing* and *Flight Controller*. It uses the sensor and remote data received from MicroBlaze$_0$ via VC3 and delivers the

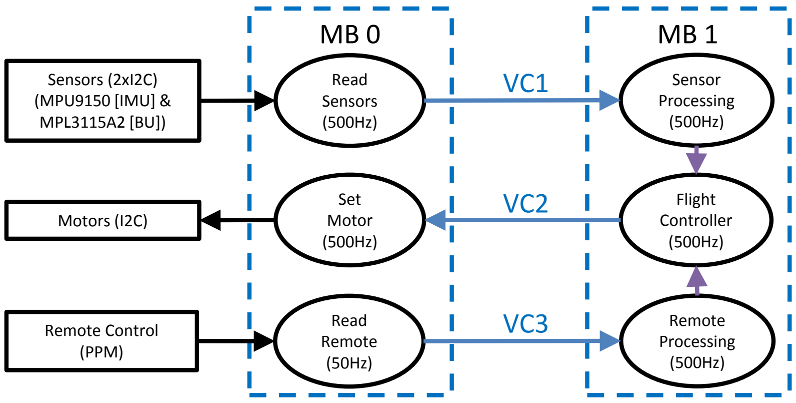

MB 0
MB 1
Sensors (2xI2C) (MPU9150 [IMU] & MPL3115A2 [BU])
Read Sensors (500Hz)
VC1
Sensor Processing (500Hz)
Motors (I2C)
Set Motor (500Hz)
VC2
Flight Controller (500Hz)
Remote Control (PPM)
Read Remote (50Hz)
VC3
Remote Processing (500Hz)

Figure 10.7.: Multirotor mapped tasks [51]

final motor set points. Virtual Channels (VC) are a TDMA bus that implements time-triggered communication through unidirectional virtual channels.

The following sections explain processing units further and present their consisting functions.

Sensor Processing

The sensor processing unit is the controller's interface to the outside world. It interfaces on the one hand with the I^2C bus to the sensors and actuators and on the other hand with the PPM interface for receiving remote controller commands.

A decomposition of the sensor actuator component is given in Figure 10.8. The sensor actuator component is implemented on a Microblaze microprocessor system like the flight controller. The Microblaze is connected to local memory, two timers for messages and tasks, an interrupt manager, and a peripheral manager. The peripheral manager allows it to access memory, General-Purpose Input/Output (GPIO), timer, I^2C bus, and PPM receiver and interpreter.

The sensor actuator partition provides the processed sensor values and PPM messages to the flight controller and receives the motor set-points values back. The motor values are then transferred to the multirotor's actuators (i.e., motors) through I^2C protocol.

As shown in Figure 10.7, There is VCs between flight controller and sensor-actuator units. The sensor actuator sends all the values as separate specific messages according to a message schedule in a time-triggered fashion to the flight controller. The

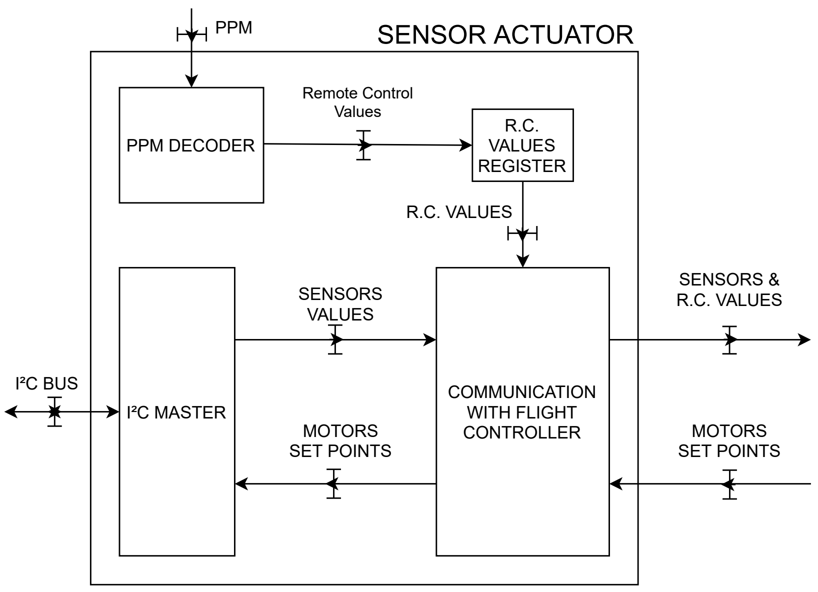


Figure 10.8.: Decomposition of sensor actuator component

time-triggered fashion specifies when a task starts when a message will be sent, and when the received message from the flight controller should be read.

The sensor processing unit has three main tasks, which are mentioned below.

Read Sensors: This component reads data from the Inertial Measurement Unit (IMU) and the Barometric Unit (BU) sensors every 2 ms and transmits the values to the local memory for sensor data processing. The sensor data include $ACC\_X/Y/Z$ for current acceleration, $GYRO\_X/Y/Z$ for current attitude, and $MAG\_X/Y/Z$ for current magnitude, current height, thrust control, yaw control, roll control, nick control. They are used to compute the attitude and the altitude.

Read Remote: This component requests the current value of the remote control every 20 ms and transmits it to the local memory for the remote control pre-processing. The remote control receiver has an update rate of 20 ms.

Send Motor Values: This last component transmits the computed motor values to the motor drivers over the provided hardware interface. Motor set-points are $REAR\_POINT$, $FRONT\_POINT$, $LEFT\_POINT$, and $RIGHT\_POINT$ (front, left, rear, and right motor points).

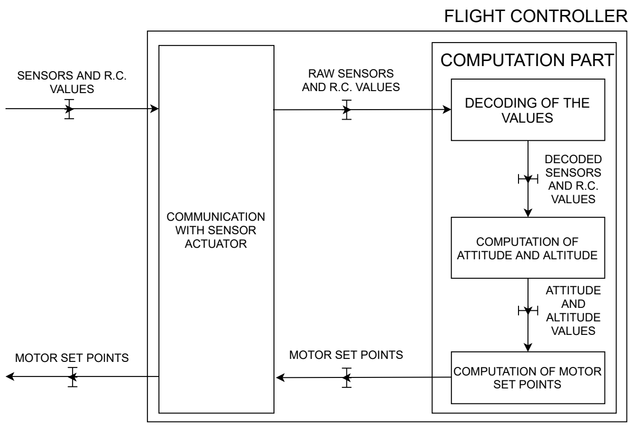

FLIGHT CONTROLLER
SENSORS AND R.C. VALUES
COMMUNICATION WITH SENSOR ACTUATOR
RAW SENSORS AND R.C. VALUES
COMPUTATION PART
DECODING OF THE VALUES
DECODED SENSORS AND R.C. VALUES
COMPUTATION OF ATTITUDE AND ALTITUDE
ATTITUDE AND ALTITUDE VALUES
COMPUTATION OF MOTOR SET POINTS
MOTOR SET POINTS
MOTOR SET POINTS

Figure 10.9.: Decomposition of Flight Controller

Flight Controller

The flight controller is the central part of our multirotor. It is connected to local memory, two timers for messages and tasks, an interrupt manager, and a peripheral manager. The peripheral manager allows it to access memory, and GPIO, and timer.

A decomposition of flight controller is given in Figure 10.9. It is another representation of the flight controller (MB1) in Figure 10.7 which has substitute function names with their simple main roles and the order of execution.

When MB1 receives the message (VC1, VC2, VC3), the values are decoded to make the raw values understandable for the controller. After decoding, the next step is to compute the attitude and altitude concerning the newly arrived inputs. Then, the computation of the motor set-points takes place. Once the calculation of the motor set-points is done, the motor set points are sent back to the sensor actuator component via the corresponding VC.

Different functions of the controller, as it was shown in Figure 10.5 and explained in the previous paragraph, can be further formalized as follow:

Sensor Processing: The available sensor values are processed to get the current attitude and altitude. In sub-components, filters are applied to smoothen the sensor values over a short time and scale the values to fit their physical units, e.g., $[°/s]$, $[m/s^2]$.

Remote Processing: Similar to the sensor values, the remote values are processed to generate the needed set points for the controllers. This component scales the remote values to fit the physical units for the set points.

Flight Controller: The Flight Controller component consists of the controller for the attitude and the altitude. The attitude controller contains three PID controllers for the nick, roll, and yaw axes. The altitude controller has one PID controller for the global Z-axis (height), with a modified integral part to avoid overshooting and undershooting of the system. All four controllers are also influenced directly by the speed given by the sensor values (for rotation axes, the values of the gyroscopes, and the height, the first integral of the acceleration along the global Z-axis). All controllers have combined position and speed controllers. This gives the system a much more stable behavior. As inputs, all controllers get their current measured values and the adjusted set points to compute the error and control the outputs. The last component in this sub-chain uses the output values of the controllers and mixes these with the four motor values.

10.2.3 Adapter Board

The board chosen for the adapter board is Zybo Zynq-7000 ARM/FPGA SoC development board as shown in Figure 10.10. It has enough capacity for the design. The Zybo is the smallest member of the Xilinx Zynq-7000 family, the Z-7010. It is based on the Xilinx All Programmable SoC architecture, which tightly integrates a dual-core ARM Cortex-A9 processor with Xilinx 7-series FPGA logic.

The adapter board converts the values from the flight simulator via Ethernet to an I^{2}C protocol and abstracts away the flight simulation to the multirotor set up as if the flight controller board is in the operational field. Instead of flying the actual physical quadcopter (Figure 10.2), with the help of the adapter board, we can use precisely the same as a HIL setup connected to a flight simulator which saves a lot of time and effort.

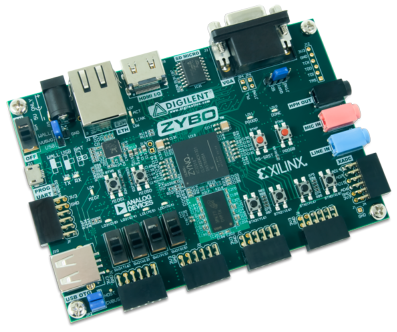


Figure 10.10.: Digilent Zybo used for adapter board

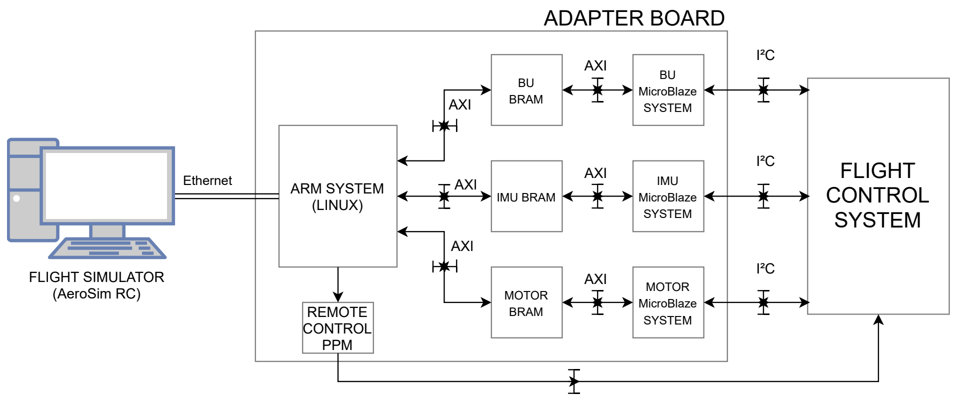


Figure 10.11.: Schematic representation of the adapter board's architecture

The Adapter board handles the I^{2}C communication between the control system and the flight simulator. It is necessary to have the adapter board because there is no other way to realize the I^{2}C communication between the sensor-actuator system of the controller and the flight simulator without altering the design and the code. Either the controller board should connect to the flight simulator via Ethernet, or a middle device is needed to translate Ethernet to I^{2}C.

A schematic representation of the adapter board is given in Figure 10.11. The adapter board uses a Microblaze microprocessor to transfers= the values inside the Block-RAM (BRAM) memory to the flight control system and vice-versa. On the ARM processor, an embedded Linux exchanges the values from the flight simulator to the BRAM and performs the necessary data conversion. The system also includes a PPM manager that sends the PPM signal (which contains the remote controls) to the flight control system. The Linux is booted from an SD card on the Zybo board. This SD card also contains the bitstream and elf binary files for the FPGA.

It is essential to note the name of the different parts of the system. The Barometric Unit (BU) represents the MPL3115A2 sensor (height and temperature), the Inertial Measurement Unit (IMU) represents all the other sensors, and the Motors represent the four different motors. All those parts simulate the behavior of the sensors as I^{2}C slaves of the flight control system's sensor actuator.

In the Xilinx IP Catalog, there is no I^{2}C IP core which works as a slave. They are designed so that users would like to use them to access an external sensor, so some work is needed to use them as a slave that acts as a sensor. The code on the Microblazes of the adapter board handles the interrupt coming from the I^{2}C components. It then, depending on the kind of sensor, whether it is output or input, reads or writes from/to BRAM.

The software running on the Linux operating system of the adapter board works as a mediator between the virtual world and the actual hardware. At the heart of the software, there is SimLink client software that exchanges the data from the AeroSIM RC flight simulator to the programmable logic of the board and from programmable logic back to the AeroSIM RC via Ethernet. This software is also responsible for initiating the ping-pong synchronization protocol by waking up the flight controller board when sensor values are ready to be sent to the controller board.

10.2.4 Virtual Platform

Software virtual platform provides a prototyping infrastructure for the software developers. It is an instruction-accurate representation of the target hardware and implements only the needed functionality. Before we test our implementation in the field, we use a VP setup to be able to debug the design easier and avoid the hardware complexity.

In software development, to run the code in an embedded system, an ISS equipped with a debugger is used. For the systems with multiple processors/cores, though, only one single ISS is not enough. What is needed is a platform model that includes models of all the processors or cores, the peripherals, and behavioral components that the software communicates with. This is what it is called a Virtual Platform. A Virtual Platform is a software-based system that can fully mirror the *functionality* of an SoC or a board. The VP abstracts all the hardware implementation details with the help of fully functional models of the hardware building blocks to provide an executable representation of the hardware to software developers and system architects.

VPs have the following advantages:

1. It enables software development at the early stage of the development of silicon, RTL simulation, or FPGA prototype availability.

2. It provides high debuggability and control over the design, which is not available or very difficult in the physical hardware. It is not possible to stop the platform and un-intrusively inspect the components at the hardware level. However, at the VP level, it is possible to stop the platform at any time and take a look at all the different parts of the system.

3. It provides determinism in the form of repeatability of test cases. The behavior is the same in each run. However, it is difficult to reproduce an intermittent problem or a timing-related problem in hardware.

4. It can be shared across different teams during development, and there is no need to wait until production. The discussions also bring an excellent common ground between software and hardware developers.

We have chosen Imperas OVP as our simulation platform. OVP virtual platform provides an environment to allow software debugging and analysis and also helps to test the system configuration independent of the hardware platform. OVP has all the features that are desired for us, such as a rich set of APIs and custom scheduling feasibility. OVP provides libraries of processor and behavioral models and APIs for building your processors, peripherals, and platforms [42].

OVPsim is a critical component of the OVP initiative. OVPsim is a multiprocessor platform emulator (full-system simulator) used to run exact production binaries of the target hardware. It uses dynamic binary translation technology to achieve high simulation speeds [42]. In the following, three essential elements in the VP are discussed better to understand the underlying VPIL infrastructure.

1. QuantumLeap

   In the simulation of the multicore platform, time moves forward in quanta. During each quantum, processors may run in parallel in independent native threads, but they are all synchronized at the quantum end before the next quantum starts.

   However, any processor may cause the simulation to revert to synchronous mode during a quantum if the simulator detects that synchronous operation is required. All other processors are safely stopped in such a case while the atomic action is carried out on the processor requiring synchronization. In other words, *quantum is the simulation step*.

   OVP uses quantum technology called *QuantumLeap* for virtual platform simulation acceleration. A parallel simulation performance accelerator provides a fast virtual platform software execution speed. It improves the simulation of multicore embedded processors System-on-a-Chip hardware platforms by executing all in parallel rather than the existing single-threaded virtual platform simulators.

QuantumLeap synchronization algorithm minimizes the impact of communication between the parallel cores, allowing them to scale across available host PC processors as much as possible.

2. MIPS

   Another parameter called MIPS is used to specify the nominal processor speed in millions of instructions per second. This nominal MIPS rate is used to apportion run time between processors in a multiprocessor simulation.

3. Interception Library

   OVP models are created using C/C++ APIs. One of the main OVP APIs is VMI which provides the processor modeling. These API functions offer the ability to describe a processor's behavior easily. A processor model written in C using the VMI decodes the target instruction to be simulated and translates this to native x86 instructions that are then executed on the PC. There is an **interception mechanism** enabling emulation of calls to functions in the application runtime libraries without requiring modification of either the processor model or the simulated application [42].

   In our implementation, we use the VMI API to intercept the code execution to enforce our code annotated scheduling. The intercept library uses the debug interface of the model and uses conditions in the code execution as a breakpoint.

   Semihosting is the term used for opaquely intercepting calls to specific functions, performing the requested work using the host system resources, and returning data as if the function were performed on the simulated system.

   With the help of the interception library, we break the code execution on the timer interrupts received from the program when tasks should be executed, or messages arrive or should be sent. Then in the interception function, we set the time and return the function to the normal execution.

Experiments 11

This section performs a flight simulation with the Hardware-in-the-the-Loop (HIL) and the Virtual Platform-in-the-Loop (VPIL) setups. The HIL experiment serves as our reference model. We evaluate our VPIL by comparing it to the reference model. We examine the accuracy of the GALI approach in comparison with other instruction-accurate simulation techniques concerning the reference model.

11.1 Hardware-in-the-Loop

The Hardware-in-the-Loop (HIL) setup, as illustrated in Figure 11.1, consists of 1) ZedBoard that implements the flight controller and sensor acquisition, 2) AeroSim RC flight simulation software which runs on a PC and provides the simulation environment and 3) Zybo board as an adapter which is connected from one side via Pmod connection to the Zedboard and from the other side via Ethernet to the flight simulation software. All elements are discussed in details in the evaluation setup (Section 10.2). Figure 11.1 depicts the experimental setup for the evaluation of the reference simulation model as the golden model for the evaluation of the GALI simulation accuracy.

The flight controller is working with a 500Hz clock frequency, and therefore, every 2ms new motor set-points will be calculated. This update rate allows the system to react quickly to environmental or set point changes. Our simulation setup can

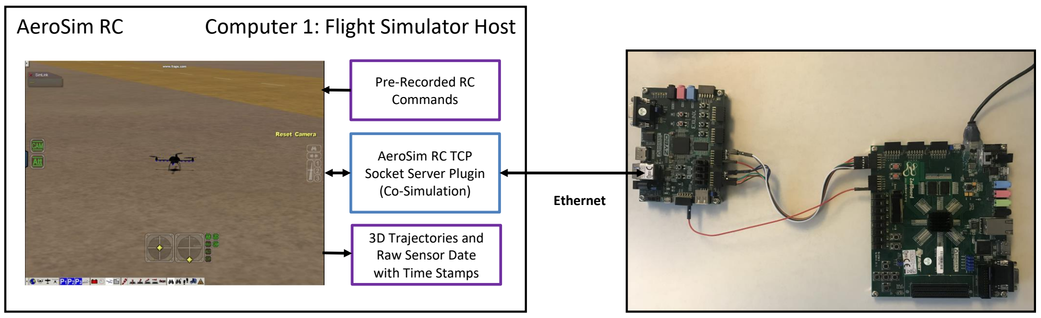

AeroSim RC
Computer 1: Flight Simulator Host
Pre-Recorded RC Commands
AeroSim RC TCP Socket Server Plugin (Co-Simulation)
3D Trajectories and Raw Sensor Date with Time Stamps
Ethernet

Figure 11.1.: HIL Setup for the Evaluation

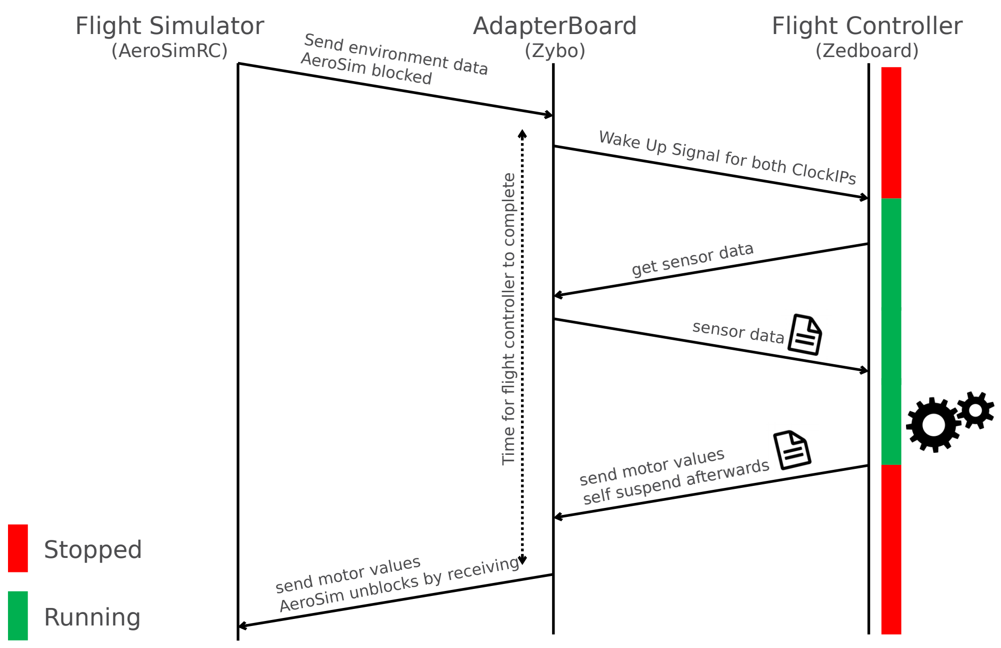


Figure 11.2.: Hardware-in-the-Loop Co-Simulation Synchronization Schema

not contend with 500Hz, and it is several orders of magnitude slower. It takes more than 40ms until the adapter board sends data and receives the reply from the PC (only the Ethernet connection part).

Hence, we implemented a synchronization mechanism on both the multirotor and adapter boards to block the simulation to ensure consistent values. Without the synchronization, the motor set-points were sent back to the flight simulator belonging to the remote commands from more than 40ms ago. This so-called *ping-pong* synchronization mechanism in the HIL is illustrated in Figure 11.2.

For implementing the *ping-pong* mechanism, the controller and the sensor processing units on the controller board (ZedBoard) are equipped with a clock gating module. It is used just for the HIL setup and is not part of the multirotor logic. It slows down the hardware board with the speed of the slowest element in the design. Although it slows down the simulation speed for synchronization, it produces correct results. Clock gating IP is a module with a clock buffer from Xilinx *BUFG* library component. BUFGCE is a general clock buffer with clock enable. It is a clock buffer with one clock input, one clock output, and a clock-enable line [52]. With the help of this component, we can freeze the ZedBoard and halt the system state to synchronize with flight simulation. At the end of each period, after executing the last task on each unit, a clock gating function is called on both task's timer and message's timer of each unit. Since the timer is frozen, the time will not proceed for the tasks, although other parts of the design, for instance, I^2C modules and the Microblaze auxiliaries, are still clocked, although they have no task to process. The system is in a semi-frozen state until a wake-up command is issued from the adapter board to waken up the controller board to read the new sensor values and calculate the new motor set-points.

On the adapter board, the flight simulator sends the environmental variables via Ethernet, while the flight controller is in the frozen mode via clock gating IP and waits for the motor values. The adapter board translates the received values from the flight simulator software via Ethernet to I^2C protocol. It then activates the clocking of the ZedBoard by sending a wake-up signal and blocks the flight simulation in a busy loop until new motor data arrive. As soon as the flight controller computes the corresponding motor set points, it sends them back to the adapter board via I^2C lines and turns its clock source off. The adapter board transfers the newly computed motor values to the flight simulator and waits until it gets the new sensor values from the simulator. It then repeats the same procedure over and over again.

To perform the experiment, pre-recorded remote control commands from a successful takeoff, hovering, and landing are logged in a `.csv` file and are fed to the simulator. As soon as the connection is established, the simulator starts reading the remote control commands from the CSV log file and turns on the engines. AeroSim RC exchanges the sensor values and remote controller commands with the motor set-points from the adapter board.

After replaying the saved log run is finished on Computer1 (Figure 11.1) in AeroSim, all the raw sensor data with the timestamps of this simulation run are dumped in another log file. The dump file contains all the necessary data to build a 3D trajectory visualization of the flight path. We take these 3D trajectories as our golden model and examine the accuracy of different virtual platform simulation scheduling techniques concerning this trajectory from the HIL setup. Figure 11.3 shows 10 execution of the HIL experiments. Differences in the flight trajectories are because of the sensor noise in the flight simulation environment.

11.2 Virtual Platform-in-the-Loop

Virtual Platform-in-the-Loop (VPIL) consists of two computers connected via Ethernet. Figure 11.4 depicts the experimental setup for the evaluation of the GALI simulation technique. On one system, *Computer1*, the AeroSim RC flight simulator is

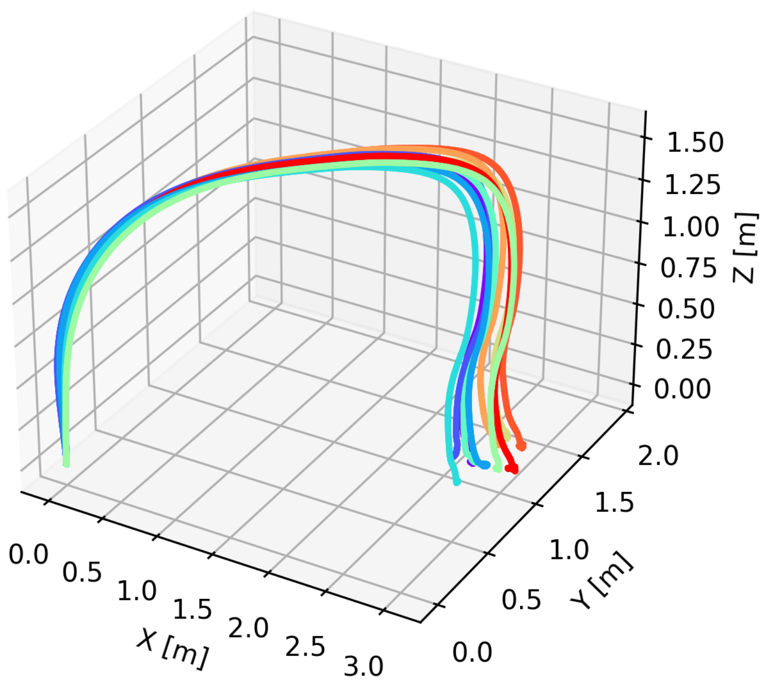
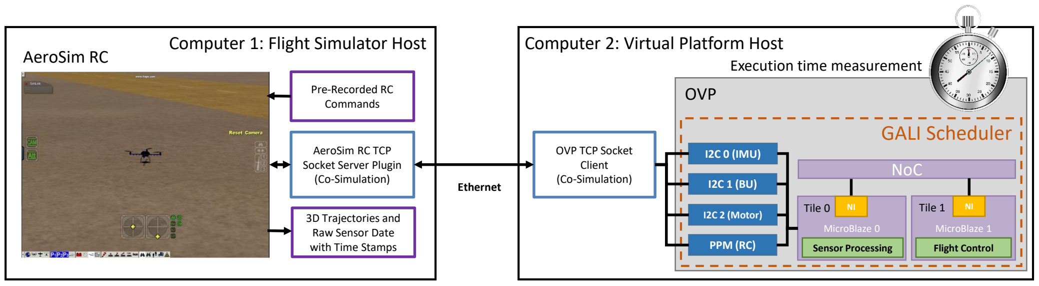


Figure 11.3.: 3D flight trajectories from HIL experiment



Computer 1: Flight Simulator Host
AeroSim RC
Pre-Recorded RC Commands
AeroSim RC TCP Socket Server Plugin (Co-Simulation)
3D Trajectories and Raw Sensor Date with Time Stamps
Ethernet
Computer 2: Virtual Platform Host
Execution time measurement
OVP
GALI Scheduler
OVP TCP Socket Client (Co-Simulation)
I2C 0 (IMU)
I2C 1 (BU)
I2C 2 (Motor)
PPM (RC)
NoC
Tile 0
NI
MicroBlaze 0
Sensor Processing
Tile 1
NI
MicroBlaze 1
Flight Control

Figure 11.4.: VPIL Setup for the Evaluation [51]

running, and on the other one, *Computer2*, the OVP virtual platform. The experiment procedure is very similar to the HIL.

The experiment procedures in HIL and VPIL are similar. The adapter board is implemented as an OVP software module (OVP TCP socket client) and initiates the connection. The socket client translates all the I^2C data from the controller board and sends it to the flight simulator. *Computer1*, as shown in VPIL setup (Figure 11.4) is exactly the same as in HIL setup (Figure 11.1).

On Computer1, the socket service plugin of the AeroSim RC is waiting for the connection to be established. Pre-recorded remote control commands from a successful takeoff, hovering, and landing are logged in a CSV file and are fed to the simulator.

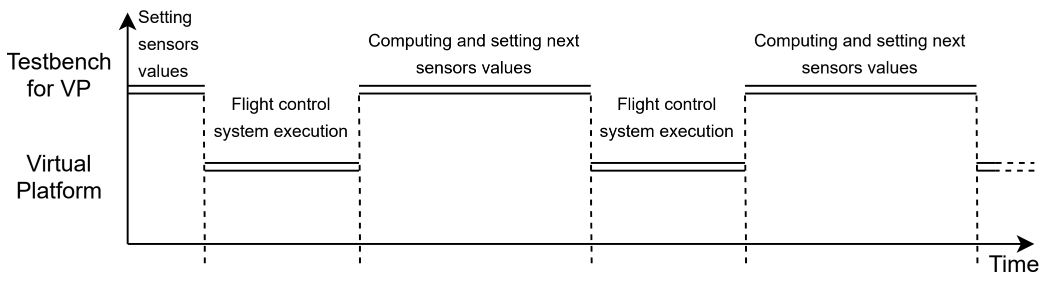

Testbench for VP
Virtual Platform
Setting sensors values
Flight control system execution
Computing and setting next sensors values
Flight control system execution
Computing and setting next sensors values
Time

Figure 11.5.: Synchronization mechanism

As soon as the connection is established, the simulator starts reading the control commands from the CSV log file and turns on the engines. AeroSim RC exchanges the sensor values and remote controller commands with the motor set-points from Computer2. The simulation is performed in a blocking fashion, which means that the simulator halts the simulation until it gets the next motor set-points and then proceeds to the next timestep. Figure 11.5 visualizes this synchronization process. Sensor values are set in the flight simulator. The virtual platform receives the environmental variables, executes the flight control system, and sends the motor values to the simulator. The flight simulator repeatedly generates the next patch of the sensor values, and the virtual platform generates the corresponding motor set points.

On computer2, the OVP TCP socket client initiates the connection. All the I^2C data reads and writes go through the socket client. In this setup, the default scheduler of the OVP is replaced with the GALI scheduler (explained in Section 8). The binaries running on OVP are labeled for GALI scheduling, and the execution order is applied based on the task timestamps. On the virtual platform, GALI scheduler replaces the task's and message's timers with labels in the source code and executes each task and sends a message based on the schedule, and sets the time corresponding to that task or message.

After replaying the saved log run is finished on Computer1 in AeroSim, we used the 3D trajectories of each flight to compare the accuracy of a flight in a virtual platform with different scheduling algorithms with a flight performed in the HIL setup.

We evaluate the viability of our approach by testing it on a fully functional safety-critical flight control system for multirotor. In the evaluation of GALI, we are only focusing on the safety-critical flight control function. In the evaluation, we are interested in the functional behavior of the different simulator configurations and the required simulation time. The following simulation models are used to compare the accuracy of state-of-the-art simulation techniques in instruction accurate simulation and to highlight the impact of the GALI:

1. **ACA model**: The Approximate Cycle Accurate (ACA) model use the OVP default timing model. In this model, each processor has a specific MIPS. We are using different MIPS rate and quantum size configurations in the experiments to demonstrate their effect on the system behavior and simulation time.

2. **QCA model**: The Quasi Cycle Accurate (QCA) model extends OVP towards a cycle-accurate model [46]. It includes a model of the processor pipeline and the memory interface.

The flight simulator records 3D flight trajectories for each simulation model mentioned above. Flight trajectory is a vector of recorded remote control data, translated to a vector of set points for the PID controllers. Furthermore, the flight simulator gets reference remote control data that defines the reference flight trajectory.

Figure 11.6 shows ten execution of flight simulation by using the GALI scheduling technique. Differences in the flight trajectories are because of the sensor noise in the simulation environment. From the look, it is very similar to the trajectories generated by HIL (shown in Figure 11.3). A detailed comparison is presented in the following chapter.

11.3 Discussion

In this section, we discuss the results obtained from the Hardware-in-the-Loop and Virtual Platform-in-the-Loop experiments and explain all the tests and procedures executed in our GALI-CE framework before and after the experiments.

First, we begin with the GALI scheduler, which shows the trace comparison and verifying of the functional correctness. Next, we compare different techniques in instruction accurate simulation concerning simulation time. The last point of the discussion is the comparison of functional accuracy. We compare the proposed GALI technique, especially with the results obtained from the HIL setup. In the end, we summarize our discussion and conclude this chapter.

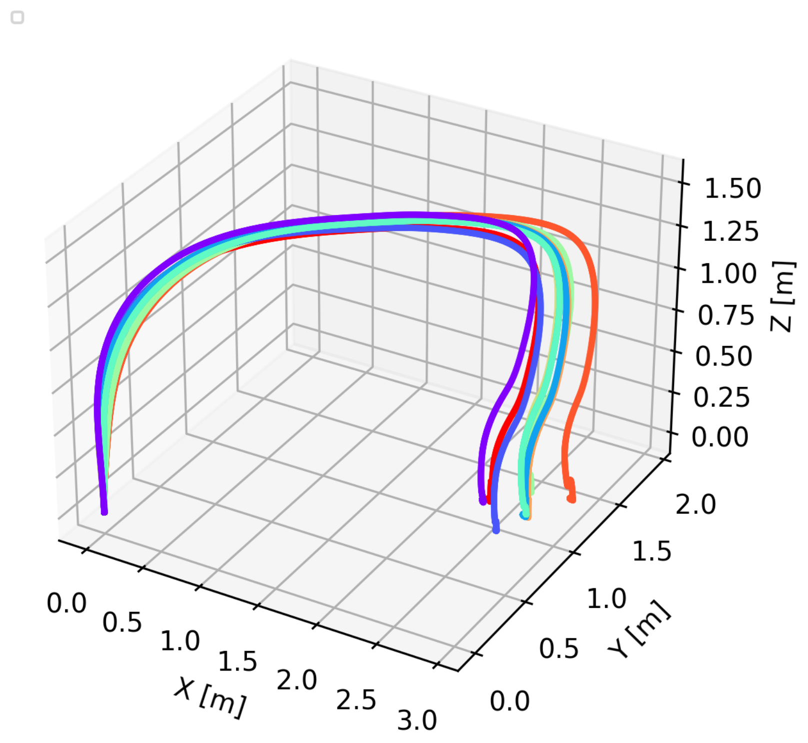


Figure 11.6.: 3D flight trajectories from GALI experiment

11.3.1 GALI Scheduler

The initial evaluation is conducted with the timed-automata for validating the system behavior by comparing the OVP trace from GALI scheduler against the UPPAAL[49] timed-automaton model. Table 11.1 shows a comparison of the traces for the sensor processing unit (MB0). The comparison of the trace for a full hyperperiod verifies the accuracy of the scheduler generator. For the task names and definitions, refer to Section 9. Table 11.1 depicts two sets of columns for TA and GALI traces. It shows iteration number 0 and number 10, just as an example. GALI trace matches precisely with the system model, and we can make sure that the implementation satisfies the definition.

11.3.2 Simulation Time

In this section, we discuss the time duration that each simulation technique needs to perform a complete flight scenario. Table 11.2 shows the comparison of different simulation times of the conducted experiments under the previously described simulation setup. Each experiment in Table 11.2 is performed under the same conditions, such as the same input trajectory and the same weather conditions.

TA			GALI	
Loc.	i	t [ms]	Nr. instr.	time [s]
Schedule	0	0	0	0
T1	0	0	445	0.00000
VC1	0	55	83	0.00055
VC3	0	55	91	0.00055
T2a	0	55	58	0.00055
Idle_T2a	0	65	0	0.00065
Idle_T2b	0	65	0	0.00065
T3	0	83	106	0.00083
Idle_T3	0	105	0	0.00105
VC2_Idle	0	105	0	0.00105
T4	0	105	3176	0.00105
Idle_T4	0	110	0	0.00110
T5	0	110	354	0.00110
T6	0	112	504	0.00112
T7	0	122	0	0.00122
T8	0	122	743	0.00122
Slack0	0	200	0	0.00200
Schedule	1	0	0	0.00200
...	...	...	...	...

TA			GALI	
Loc.	i	t [ms]	Nr. instr.	time [s]
Schedule	10	0	0	0.02000
T1	10	0	445	0.02000
VC1	10	55	83	0.02055
VC3	10	55	91	0.02055
T2a	10	65	58	0.02065
Idle_T2a	10	65	0	0.02065
Idle_T2b	10	83	0	0.02083
T3	10	87	84	0.02087
VC2	10	87	86	0.02087
T3	10	87	106	0.02087
Ilde_T3	10	105	0	0.02105
T4	10	110	3176	0.02110
Idle_T4	10	110	0	0.02110
T5	10	112	354	0.02112
T6	10	122	504	0.02122
T7	10	122	0	0.02122
T8	10	200	743	0.02200
Slack0	10	200	0	0.02200
...	...	...	...	...

Table 11.1.: Excerpt of the Timed Automata (TA) and GALI trace for MB0
The left side shows the values for the iteration Nr. 0 and the right side for iteration Nr. 10. The values in between are skipped.
Loc.: Location on the timed-automata, *i*: number of current iteration on a period, *Nr. instr*: Number of instruction executed for the task (because of semi-hosting this number has been extremely decrease.

The only non-controllable input variable is the sensor noise. For this reason, the simulator has been repeated ten times on the simulation host. The average duration of these 10 measurements is shown in the Table 11.2 column "avg. duration". The simulation time is the time spent on CPU[1] in the virtual platform host (Computer2 in Figure 11.4) for running the OVP simulator. This time has been measured with `perf`, a performance analyzing profiling tool with performance counters in Linux and capable of statistical profiling of the entire system. The command used to measure the simulation time:

```
perf stat -e cpu-clock -r 10 <OVP Simulator>
```

This experiment has two parameters to adjust: MIPS rate and Quantum size. First column in Table 11.2 is the MIPS rate. In IA simulators like OVP, there is a *concept* of time, but they do not accurately model processor speed – one needs clock-cycle

[1]Intel i7-4710MQ(2.5GHz) with 16GB RAM running Debian Jessie (Kernel 4.12.08 x86 64).

Experiments				Simulation Time			
No.	Model	MIPS rate	Quantum size [s]	avg. duration [s]	error [%]	speed-up w.r.t. QCA	wall clock rate
1	QCA	-	-	121.585	0.19	1.0	0.09
2	ACA/IA	1	0.0001	2.374	0.14	51.2	4.38
3	ACA/IA	10	0.0001	2.192	0.17	55.2	4.74
4	ACA/IA	100	0.0000001	35.925	0.30	3.4	0.29
5	ACA/IA	100	0.0001	2.032	0.25	59.8	5.12
6	ACA/IA	100	0.01	1.959	0.27	62.1	5.31
7	ACA/IA	1000	0.0001	11.788	0.27	10.3	0.88
8	ACA/IA	79	0.0001	1.785	0.32	68.1	5.83
9	GALI	-	-	0.758	0.24	160.4	13.72

Table 11.2.: Experimental results - Comparing different simulation models in virtual platform-in-the-loop.
QCA: Quasi Cycle Accurate.
ACA/IA: Approximate Cycle Accurate Instruction Accurate.
MIPS: Million Instructions Per Second.
Simulated time is used to compare the simulations, but will not give an accurate indication of speed in the real hardware.

accuracy and MHz clock to do this. The simulation time is advanced at the end of each simulation quantum (or simulation step). The number of executed instructions in each quantum is specified by setting the MIPS rating parameter. The specified MIPS (Million Instructions Per Second) parameter value specifies the number of instructions to be executed in a measure of elapsed real-time. OVP processor models have a default speed of 100 MIPS. MIPS is a measure of processor speed which should not be confused with MIPS Technologies Inc., a processor IP vendor.

All the timing models in instruction accurate simulation rely upon counting the executed instructions. The nominal processor speed MIPS rate is used to apportion run time between processors in a multiprocessor simulation. This rate is the average amount of instructions that the processor can execute in one second and quantum is the maximum amount of simulated target instructions per processor. Quantum defines the timing granularity for switching, and thus synchronization, between the processors under simulation.

The GALI model is not dependent on any parameters of the simulator, including MIPS. It is therefore faster than other models (looking at row 4 to row 8 in Table 11.2 - about $2.3\times$ up to $47\times$ faster). GALI is also 160 times faster than the QCA model. It all concludes with a significant advantage for GALI in the simulation speed. This gain in simulation speed depends mainly on the fewer context switching in the simulator. In our experiment, we have assigned different MIPS rate and quantum size values to show their influence on the simulation speed.

Experiments				Functional Accuracy			
No.	Model	MIPS rate	Quantum size [s]	max. σ_x	max. σ_y	max. σ_z	avg. rel. error [%]
1	HIL	-	-	0.14	0.14	0.02	-
2	QCA	-	-	0.14	0.27	0.04	0.24
3	ACA/IA	1	0.0001	0.10	0.16	0.05	92.16
4	ACA/IA	10	0.0001	0.18	0.27	0.03	3.56
5	ACA/IA	100	0.0000001	0.16	0.15	0.02	1.09
6	ACA/IA	100	0.0001	0.18	0.19	0.04	1.29
7	ACA/IA	100	0.01	0.15	0.13	0.04	2.87
8	ACA/IA	1000	0.0001	0.22	0.19	0.04	1.31
9	ACA/IA	79	0.0001	0.12	0.25	0.04	2.16
10	GALI	-	-	0.17	0.14	0.03	0.70

Table 11.3.: Experimental results - Accuracy

The speed-up factor in Table 11.2 compares the simulation time of each experiment with the simulation time of the QCA model[46] which was the most precise instruction accurate timing model available to us. A speed-up factor of x indicates that the simulation of the respective model runs x times faster than the QCA model. The reference QCA experiment, for the measurement of the simulation time, takes ~ 10.4 seconds when running freely (i.e., without co-simulation)[2]. Under this condition, the GALI model reaches 13.7 times wall clock time, while the ACA model might fail to run $\leq$ one wall-clock time, which can be necessary for a real-time capable co-simulation.

11.3.3 Functional Accuracy

A more critical factor than the speed-up factor is the accuracy of the respective model. Table 11.3 shows the results of the conducted experiments under the described simulator set up with the focus on the functional accuracy. In the evaluation, we are interested in the functional behavior of the different simulator configurations. We have calculated the maximum standard deviation in each experiment in all three directions (max. σ_x, max. σ_y, and max. σ_z).

As mentioned before, the MIPS rate is the average amount of instructions that the processor can execute in one second and quantum is the maximum amount of simulated target instructions per processor. Quantum defines the timing granularity for switching, and thus synchronization, between the processors under simulation.

[2]Applications run ca. 10 Seconds (5000 controller cycles) in real-world plus init phase (overall: ca. 10.4s).

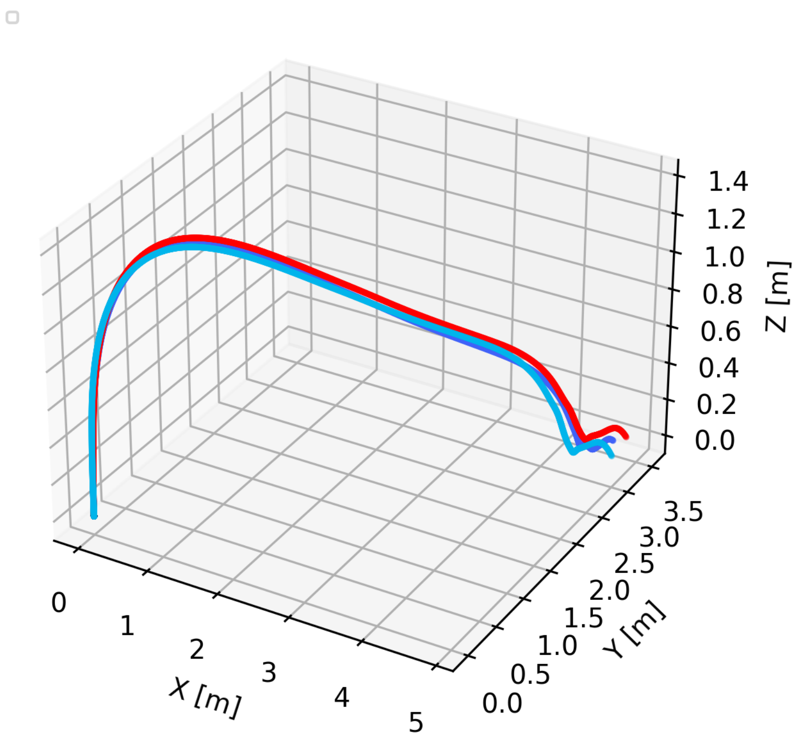
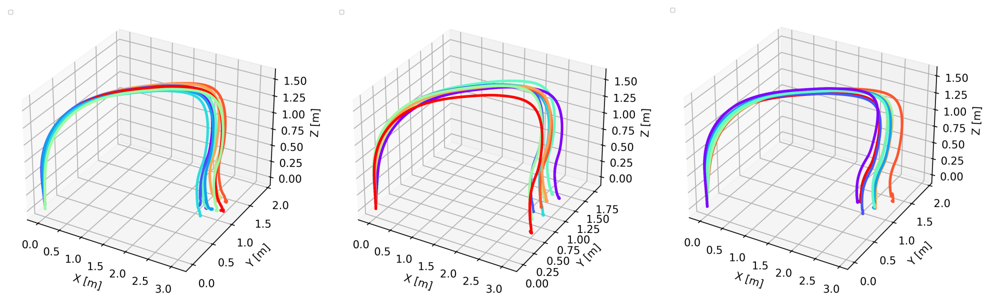


Figure 11.7.: Flight crash due to mis-configuration of the virtual platform

The functional accuracy of the different simulation models is measured against the reference flight trajectory of the HIL. Functional accuracy assessment is always use-case dependent. Our multirotor use case has a very robust PID control loop, which is very agnostic to disturbances and environmental noise. Therefore, the accuracy advantage of GALI compared to the ACA and QCA experiments is not too significant. Much higher deviations would be observable with another use case with a less robust control system. The reason for highly achievable functional accuracy for the ACA models is the robustness of the PID controllers that the controllers still work well under high under-sampling.

By construction, it should be clear that GALI represents an ideal time-triggered schedule. For this reason, the obtained functional accuracy compared to a “golden” functional reference model should be 100%. In our experiments, we have to deal with non-controllable sensor noise. For this reason, GALI was not able to reach complete accuracy, even though the PID controllers are very robust to noise.

The average relative error in the Table 11.3 is the difference of the mean QCA, ACA and GALI trajectories from the mean HIL reference trajectory. Average relative accuracy is the average of the X, Y, and Z-axis, the average accuracy deviation divided by the average reference trajectory for each axis. The standard deviations



(a) HIL **(b)** QCA **(c)** GALI

Figure 11.8.: Selected 3D flight trajectories from functional accuracy experiments

(σ_x, σ_y, and σ_z) represent the reaction of the PID controllers to the non-controllable sensor noise within the repeated flights within a single experiment. The average relative error is the deviation between the reference experiment (Nr. 1 HIL) and the other experiments (2-10).

When comparing the average relative error between the different experiments, it becomes clear that ACA models with appropriately chosen MIPS rates can lead to a sufficient functional accuracy. As expected, the GALI model has the lowest average relative error after QCA (0.70 and 0.24 respectively in Table 11.3). GALI does not depend on parameters like MIPS or quantum. Therefore there is no way that a misconfiguration of GALI leads to a crash scenario. Figure 11.7 is a ACA model with the wrong configuration, which leads to a crash landing of the multirotor.

Figure 11.8 depicts the 3D flight trajectories of all ten experiments in a single figure (time runs from left to right). It shows 3D flight trajectories for the HIL (reference trajectories), QCA model as a correct instruction accurate model available to us, and the GALI model. In all three models, the sensor noise has caused a drift of the multirotor, which becomes more significant as time proceeds.

11.3.4 Summary

In this chapter, we have evaluated the GALI simulation technique and its implementation framework (GALI-CE) proposed in this thesis. First, Section 9 verified the generated GALI schedule. The verification considers the custom scheduler generated for the virtual platform by the GALI-CE with a UPPAAL representation of the system schedule. It compares if the trace of execution of the hyperperiod for both representations match each other. When the generated GALI succeeds in validation, we conclude the correctness of the GALI-CE configuration environment (from Front-End to the Back-End as described in Section 9) and proceed to the next step of the evaluation.

Next, we have performed a Hardware-in-the-Loop (HIL) experiment to generate a reference model for evaluating and comparing different simulation techniques. The HIL setup consists of an FPGA board implementing the multirotor logic, a flight simulator for simulating the multirotor and its flying environment, and an adapter board to convert data between the controller board and the flight simulator.

After that, we used the same scenario and remote control commands to repeat the experiment with the Virtual Platform-in-the-Loop (VPIL). In the VPIL experiment, another PC replaces the controller and the adapter board with a software-based virtual platform representation. It has been done for different simulation configurations, including ACA, QCA, and GALI. All the flight trajectories and motor values have been recorded to be compared with one from the HIL setup.

In this chapter, we have compared the accuracy of different simulation techniques with the HIL experiment and the duration and speed-up factor of other simulations to understand better the pros and cons of the proposed new approach.

The evaluation of the proposed approach in this thesis has shown us that in the area of time-triggered systems, GALI outperforms better than typical ACA virtual platform scheduler in both simulation time as well as functional accuracy. However, the QCA scheduling shows better accuracy, and the reason is, as we move towards cycle-accurate simulation, the accuracy improves. Albeit, GALI brings acceptable accuracy together with 160 times speed-up in comparison with QCA technique.

Conclusion & Outlook 12

12.1 Conclusion

For the functional testing of a time-triggered mixed-criticality real-time control system, the following properties have to be considered: a) *Timeliness* means that the occurrence of the contractually defined function/task activation and communication injection times are fulfilled. b) *Functional Behavior* means that the behavioral response of the control systems meets the contractually agreed functional accuracy concerning the plant model control task.

For the first time, the presented work decouples the time-triggered schedule refinement from the functional behavior refinement. With the GALI simulation model, we have defined a universal functional simulation scheduler that executed time-triggered schedules. GALI can be embedded into instruction accurate simulators to trigger instruction-accurate functional simulation between the time-triggered function/task activation.

The evaluation of the GALI approach has shown that it has similar functional accuracy as a cycle-accurate simulation but with a significant speed-up factor of 160. From the functional testing perspective, an instruction accurate and a cycle-accurate simulator can also be used for verification. The main advantage of the GALI model is a Correct-by-Construction simulation scheduling configuration and its high simulation speed. On the downside, GALI only allows functional testing that relies on a pre-analyzed and confirmed schedule for a specific hardware architecture.

12.2 Future Work

Based on the method developed in this thesis, the most relevant extensions that can be addressed in the future work are the following:

Profiling Information of the System

By having a simulator for a generic time-triggered structural and behavioral modeling at the early architectural design phase that supports extra-functional analysis and complies with the time-triggered model of computation (MoC), one can extract the profiling information for better system design.

The instruction-accurate simulator can access and analyze the executed instructions (types, operands, etc.). This information can be combined with a power model for **Power Modeling Using Profiling Information** to obtain the power consumption and temperature of the system.

Profiling information can be used for further issues. Counting the instructions provides performance profiling by analyzing the number of arithmetic operations (e.g., multiplications, additions, divisions, etc.). For the developed algorithm, it can be used for **Performance Optimization Using Profiling Information**.

Also, the developed approach and its design environment provide a platform to change the design to add custom hardware or instructions to, for instance, test a hardware acceleration module. It can be done independent of GALI or its configuration environment, but it comes with the GALI-CE for free.

Simulation-based Safety Assessment of the Platform

Our focus in this work was just checking the functionality and timing. GALI-CE can be extended to generate a safe architecture as well. For example, it can cause a safety-hardened architecture with the fault-tolerant Triple Modular Redundancy (TMR) technique.

Considering our multirotor example, a safety-hardened multirotor makes the scheduling more complex by adding more functionalities to the architecture. The functionalities include modules to observe the added redundancies and voters.

From the GALI perspective, only minor changes are needed. The generated platform can inject faults into the system in a systematic and reproducible manner to test the system's safety both in software and hardware.

Thanks to reliance on low-level instruction-set simulators, faults can be injected as low-level glitches on processors. It can be combined with safety countermeasures on the binary level. The system behavior then can be checked in case of failure or fault containment properties. Furthermore, fault injection in all steps of the development process of the system can be introduced, as recommended by the IEC-61508 safety standard.

Support Dynamic Scheduling

Fix scheduling in safety-related mixed-critical systems results in a massive waste of resources. At the same time, it stays on the safe side and reserves plenty of time and resources for each task. As a result, it uses only a fraction of the computation capacity of the platform. This thesis can be extended to support flexible time-triggered scheduling. Similar to work in [53] which addresses flexible time-triggered scheduling on mono processors, single-core configuration. The work in [53] considers a mixed-criticality system with High-critical and Low-critical tasks and tries to use the slack time from high-critical tasks on the processors and performs low-critical, not real-time tasks.

Our approach can also be further developed for mixed-computation systems to have a hybrid GALI part and a regular computation part with a timing model. The configuration environment should be aware of such systems to get more information about the actual execution time of functions/tasks. The report can be used for dynamic slack-time management to fill the computation gap with appropriate low-critical tasks.

System Mode switching

GALI can be extended, similar to the previously mentioned dynamic scheduling, to support changing the system schedule on the fly. Based on predetermined scenarios and situations, systems can reconfigure themselves. One obvious case is the moment when the system goes to the degraded safe mode caused by a fault.

Mode switching of the system is more complex than dynamic scheduling. In dynamic scheduling, tasks and their resources are fixed. The challenge is to change the order of execution to improve system performance. However, in mode switching, the system would reconfigure itself. Each mode has different resource management, different task criticality, different execution order, etc. GALI is capable of handling all these situations.

Part IV

Appendix

Bibliography

[1] Hermann Kopetz and Roman Nossal. "Temporal firewalls in large distributed real-time systems". In: *Proceedings of the Sixth IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. IEEE. 1997, pp. 310–315.

[2] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[3] Hermann Kopetz. "The time-triggered model of computation". In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, pp. 168–177.

[4] Hermann Kopetz. "Elementary versus composite interfaces in distributed real-time systems". In: *Proceedings. Fourth International Symposium on Autonomous Decentralized Systems.-Integration of Heterogeneous Systems-*. IEEE. 1999, pp. 26–33.

[5] Inc. Insights. *Transistor Count Trends Continue to Track with Moore's Law*. 2020. URL: `https://www.icinsights.com/news/bulletins/Transistor-Count-Trends-Continue-To-Track-With-Moores-Law/`.

[6] Maher Fakih, Alina Lenz, Mikel Azkarate-Askasua, Javier Coronel, Alfons Crespo, Simon Davidmann, Juan Carlos Diaz Garcia, Nera González Romero, Kim Grüttner, Sören Schreiner, et al. "SAFEPOWER project: Architecture for safe and power-efficient mixed-criticality systems". In: *Microprocessors and Microsystems* 52 (2017), pp. 89–105.

[7] D.D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer US, 2009. ISBN: 9781441905048. URL: `https://books.google.com/books?id=XYZAAAAAQBAJ`.

[8] Prashant Saxena, Noel Menezes, Pasquale Cocchini, and Desmond A Kirkpatrick. "The scaling challenge: can correct-by-construction design help?" In: *Proceedings of the 2003 international symposium on Physical design*. 2003, pp. 51–58.

[9] Roopak Sinha, Parthasarathi Roop, and Samik Basu. *Correct-by-construction approaches for SoC design*. Springer, 2014.

[10] Trevor Meyerowitz. *Transaction-Level Modeling Definitions and Approximations*. Tech. rep. Technical Report EE290A, 2005.

[11] Wikipedia. *Instruction set simulator*. 2020. URL: `https://en.wikipedia.org/wiki/Instruction_set_simulator`.

[12] Joshua J Yi, Sreekumar V Kodakara, Resit Sendag, David J Lilja, and Douglas M Hawkins. "Characterizing and comparing prevailing simulation techniques". In: *11th International Symposium on High-Performance Computer Architecture*. IEEE. 2005, pp. 266–277.

[13] Roman Obermaisser, Christian El Salloum, Bernhard Huber, and Hermann Kopetz. "The time-triggered system-on-a-chip architecture". In: *2008 IEEE International Symposium on Industrial Electronics*. IEEE. 2008, pp. 1941–1947.

[14] Hermann Kopetz and Günther Bauer. "The time-triggered architecture". In: *Proceedings of the IEEE* 91.1 (Jan. 2003), pp. 112–126.

[15] Bernd Hedenetz. "A development framework for ultra-dependable automotive systems based on a time-triggered architecture". In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, pp. 358–367.

[16] Hermann Kopetz. "The time-triggered model of computation". In: *Proceedings 19th IEEE Real-Time Systems Symposium*. IEEE. 1998, pp. 168–177.

[17] Hermann Kopetz and Günter Grunsteidl. "TTP-A time-triggered protocol for fault-tolerant real-time systems". In: *FTCS-23 The Twenty-Third International Symposium on Fault-Tolerant Computing*. IEEE. 1993, pp. 524–533.

[18] John Rushby. "An overview of formal verification for the time-triggered architecture". In: *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer. 2002, pp. 83–105.

[19] Mirko Loghi, Tiziana Margaria, Graziano Pravadelli, and Bernhard Steffen. "Dynamic and formal verification of embedded systems: A comparative survey". In: *International Journal of Parallel Programming* 33.6 (2005), pp. 585–611.

[20] Vicent Brocal, Miguel Masmano, Ismael Ripoll, Alfons Crespo, Patricia Balbastre, and Jean-Jacques Metge. "Xoncrete: a scheduling tool for partitioned real-time systems". In: *ERTS2, Embedded Real-Time Software and Systems* (May 2010). URL: `https://hal.archives-ouvertes.fr/hal-02264388`.

[21] Thomas Carle, Dumitru Potop-Butucaru, Yves Sorel, and David Lesens. "From dataflow specification to multiprocessor partitioned time-triggered real-time implementation". In: *Leibniz Transactions on Embedded Systems* 2.2 (2015), 01–1-01:30.

[22] Frank Singhoff, Jérôme Legrand, Laurent Nana, and Lionel Marcé. “Cheddar: a flexible real time scheduling framework”. In: *ACM SIGAda Ada Letters*. Vol. 24. 4. ACM. 2004, pp. 1–8.

[23] M. Gonzalez Harbour, J. J. Gutierrez Garcia, J. C. Palencia Gutierrez, and J. M. Drake Moyano. “MAST: Modeling and analysis suite for real time applications”. In: *Proceedings 13th Euromicro Conference on Real-Time Systems*. June 2001, pp. 125–134.

[24] Tri-Pacific. *Rapid-RMA: The Art of Modeling Real-Time Systems*. 2003. URL: `http://www.tripac.com/htmal/prod-fact-rrm.html`.

[25] Raul Adrian Gorcitz, Thomas Carle, David Lesens, David Monchaux, Dumitru Potop-Butucaru, and Yves Sorel. “Automatic implementation of TTEthernet-based time-triggered avionics applications”. In: *DASIA*. Barcelone, Spain, May 2015.

[26] Jorgiano Vidal, Florent De Lamotte, Guy Gogniat, Philippe Soulard, and Jean-Philippe Diguet. “A co-design approach for embedded system modeling and code generation with UML and MARTE”. In: *2009 Design, Automation & Test in Europe Conference & Exhibition*. IEEE. 2009, pp. 226–231.

[27] Fernando Herrera, Héctor Posadas, Pablo Penil, Eugenio Villar, Francisco Ferrero, Raúl Valencia, and Gianluca Palermo. “The COMPLEX methodology for UML/MARTE Modeling and design space exploration of embedded systems”. In: *Journal of Systems Architecture* 60.1 (2014), pp. 55–78.

[28] K. D. Nguyen, P. S. Thiagarajan, and W. Wong. “A UML-Based Design Framework for Time-Triggered Applications”. In: *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. Dec. 2007, pp. 39–48. DOI: `10.1109/RTSS.2007.18`.

[29] Iban Ayestaran, Carlos F Nicolas, Jon Perez, Asier Larrucea, and Peter Puschner. “A novel modeling framework for time-triggered safety-critical embedded systems”. In: *Specification and Design Languages (FDL), 2014 Forum on*. Vol. 978. IEEE. 2014, pp. 1–8.

[30] Gabriel Leen and Donal Heffernan. “Modeling and verification of a time-triggered networking protocol”. In: *International Conference on Networking, International Conference on Systems and International Conference on Mobile Communications and Learning Technologies (ICNICONSMCL’06)*. IEEE. 2006, pp. 178–178.

[31] Bernd Hedenetz. “A development framework for ultra-dependable automotive systems based on a time-triggered architecture”. In: *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*. IEEE. 1998, pp. 358–367.

[32] Kyoung-Soo We, Seunggon Kim, Wonseok Lee, and Chang-Gun Lee. “Functionally and Temporally Correct Simulation of Cyber-Systems for Automotive Systems”. In: *Real-Time Systems Symposium (RTSS), 2017 IEEE*. IEEE. 2017, pp. 68–79.

[33] Gunar Schirner and Rainer Domer. “Introducing Preemptive Scheduling in Abstract RTOS Models using Result Oriented Modeling”. In: *2008 Design, Automation and Test in Europe*. Mar. 2008, pp. 122–127.

[34] Jon Perez, Carlos Fernando Nicolas, Roman Obermaisser, and Christian El Salloum. “Modeling time-triggered architecture based real-time systems using SystemC”. In: *System Specification and Design Languages*. Springer, 2012, pp. 123–141.

[35] Iban Ayestaran et al. “Modeling logical execution time based safety-critical embedded systems in SystemC”. In: *Embedded Computing (MECO), 2014 3rd Mediterranean Conference on*. IEEE. 2014, pp. 77–80.

[36] Zaher Owda and Roman Obermaisser. “Trace-based simulation framework combining message-based and shared-memory interactions in a time-triggered platform”. In: *Event-based Control, Communication, and Signal Processing (EBCCSP), 2015 International Conference on*. IEEE. 2015, pp. 1–8.

[37] Moisés Urbina, Zaher Owda, and Roman Obermaisser. “Simulation environment based on systemc and veos for multi-core processors with virtual autosar ecus”. In: *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*. IEEE. 2015, pp. 1843–1852.

[38] Zhenkai Zhang, Joseph Porter, Emeka Eyisi, Gabor Karsai, Xenofon Koutsoukos, and Janos Sztipanovits. “Co-simulation framework for design of time-triggered cyber physical systems”. In: *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. 2013, pp. 119–128.

[39] Wilfried Steiner, Günther Bauer, Brendan Hall, Michael Paulitsch, and Srivatsan Varadarajan. “TTEthernet dataflow concept”. In: *2009 Eighth IEEE International Symposium on Network Computing and Applications*. IEEE. 2009, pp. 319–322.

[40] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. “Simics: A full system simulation platform”. In: *Computer* 35.2 (2002), pp. 50–58.

[41] F. Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[42] Imperas Ltd. *Open Virtual Platforms (OVP)*. 2020. URL: `http://www.ovpworld.org/`.

[43] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. “The gem5 simulator”. In: *ACM SIGARCH Computer Architecture News* 39.2 (2011), pp. 1–7.

[44] Zaher Owda, Mohammed Abuteir, and Roman Obermaisser. “Co-simulation framework for networked multi-core chips with interleaving discrete event simulation tools”. In: *Emerging Technologies & Factory Automation (ETFA), 2015 IEEE 20th Conference on*. IEEE. 2015, pp. 1–8.

[45] Daniel Sanchez and Christos Kozyrakis. “ZSim: fast and accurate microarchitectural simulation of thousand-core systems”. In: *ACM SIGARCH Computer Architecture News*. Vol. 41. ACM. 2013, pp. 475–486.

[46] Sören Schreiner, Ralph Görgen, Kim Grüttner, and Wolfgang Nebel. “A quasi-cycle accurate timing model for binary translation based instruction set simulators”. In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. July 2016, pp. 348–353.

[47] Maher Fakih, Kim Grüttner, Sören Schreiner, Razi Seyyedi, Mikel Azkarate-Askasua, Peio Onaindia, Tomaso Poggi, Nera González Romero, Elena Quesada Gonzalez, Timmy Sundström, et al. “Experimental evaluation of SAFEPOWER architecture for safe and power-efficient mixed-criticality systems”. In: *Journal of Low Power Electronics and Applications* 9.1 (2019), p. 12.

[48] Jan-Henrik Bruhn, Maher Fakih, Kim Grüttner, and Wolfgang Nebel. “Implementation of Time-Triggered MPSoCs with support for tool-based system generation on FPGAs”. In: *4th International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems (AISTECS)*. Jan. 2019.

[49] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. “UPPAAL – A tool suite for automatic verification of real-time systems”. In: (1995), pp. 232–243.

[50] AeroSIM-RC. *Radio Control Training Simulator*. 2021. URL: `http://www.aerosimrc.com/en/home.htm`.

[51] Consortium SAFEPOWER. *D4.6 Final cross-domain public demonstrator*. 2017.

[52] Xilinx inc. *7 Series FPGAs Clocking Resources User Guide*.

[53] Philipp Ittershagen. "Application modelling and performance estimation of mixed-critical embedded systems". PhD thesis. Universität Oldenburg, 2018.

List of Figures

List of Tables

Listings

Acronyms

ACA	Approximate Cycle Accurate
API	Application Programming Interface
BCET	Best-Case Execution Time
BRAM	Block-RAM
BU	Barometric Unit
CA	Cycle Accurate
CPS	Cyber-Physical System
DLL	Dynamic-Link Library
DUT	Design under Test
EDA	Electronic Design Automation
EMF	Eclipse Modeling Framework
FPGA	Field-Programmable Gate Array
GALI	Globally Accurate Locally Inaccurate
GALI-CE	GALI Configuration Environment
GPIO	General-Purpose Input/Output
HDL	Hardware Description Language
HW	Hardware
HIL	Hardware-in-the-Loop
HDL	Hardware Description Language
IA	Instruction Accurate
IC	Inintegrated Circuit
ID	identifier
IMU	Inertial Measurement Unit
IP	Intellectual Property
ISS	Instruction-Set Simulator
MoC	Model of Computation
MPSoC	Multi-Processor System-on-a-Chip
MIPS	Million Instructions Per Second
NoC	Network-on-Chip
OVP	Open Virtual Platform
PC	Personal Computer
PE	Processing Element
PID	Proportional Integral Derivative

PPM	Pulse-Position Modulation
QCA	Quasi Cycle Accurate
RC	Radio Control
RTL	Register-Transfer Level
SDF	Synchronous Data Flow
SoC	System-on-a-Chip
SW	Software
TA	Timed Automata
TDMA	Time-Division Multiple Access
TMR	Triple Modular Redundancy
TSDF	Timed Synchronous Dataflow
TT	Time-Triggered
TTA	Time-Triggered Architecture
TTP	Time-Triggered Protocol
UML	Unified Modeling Language
VC	Virtual Channel
VP	Virtual Platform
VPIL	Virtual Platform-in-the-Loop
WCET	Worst-Case Execution Time
WCRT	Worst-Case Response Time

Declaration/Erklärung

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis an der Carl von Ossietzky Universität Oldenburg und den DFG-Richtlinien festgelegt sind, befolgt habe. Des Weiteren habe ich im Zusammenhang mit dem Promotionsvorhaben keine kommerziellen Vermittlungs- oder Beratungsdienste in Anspruch genommen.

Oldenburg,

Razi Seyyedi